



Curs pentru specializarea  
Informatică anul II  
Semestrul II,  
Horea Oros, [horea.oros@gmail.com](mailto:horea.oros@gmail.com)  
Universitatea din Oradea, Departamentul de Matematică și Informatică


# SISTEME DE GESTIUNE A BAZELOR DE DATE

# Agenda

- Recapitulare instrucțiuni SQL
- Continuarea discuțiilor despre limbajul SQL (T-SQL)
- Tranzacții
- Proceduri stocate
- Funcții definite de utilizator
- Tipuri definite de utilizator
- Trigger
- Tratarea erorilor
- Procesare condiționată, controlul execuției
- Indexare
- DCL
- Obiecte securabile, permisiuni, audit



# Tranzacții

- Sunt parte integrantă a sistemelor de gestiune a bazelor de date relaționale
  - Ajută la definirea unei singure unități de execuție
  - O tranzacție poate conține una sau mai multe instrucțiuni SQL care sunt executate sau anulate ca un tot unitar
  - Aceasta ajută la prevenirea unor actualizări parțiale sau a unor date în stare inconsistentă
- 

# Tranzacții - ACID

- Actualizare parțială = o parte a unui proces este anulat fără a anula toate celelalte părți ale aceluiași proces
- Tranzacțiile respectă regula **ACID**
  - **A**tomicity
  - **C**onsistency
  - **I**solation/Independence
  - **D**urability

# Tranzacții - ACID

- *Atomicity – tranzacțiile sunt o entitate de tip "totul sau nimic" – făcînduse toți pașii sau nici unul*
- *Consistency - asigură faptul că datele sunt valide atât înainte cât și după tranzacție. Integritatea datelor trebuie păstrată și structura internă a datelor trebuie să fie într-o stare validă*

# Tranzacții - ACID

- *Isolation - o tranzacție nu trebuie să fie dependentă de o altă tranzacție care poate să aibă loc în mod concurent. O tranzacție nu trebuie să vadă datele altei tranzacții ce este într-o stare inter mediară. Trebuie să vadă datele așa cum au fost înainte de începerea tranzacției sau după încheierea acesteia.*
- *Durability – efectele tranzacției sunt fixate după ce tranzacția este executată, și orice modificări vor putea fi anulate după un eșec al sistemului*

# Tranzacții – SQL Server

- Sunt trei tipuri de tranzacții
  - Autocommit - implicit
  - Explicit
  - Implicit
- Autocommit – fiecare comandă SQL este o tranzacție de sine stătătoare care este executată imediat
- Ex. două comenzi INSERT pentru debitarea/creditarea unui cont
- Ex. ștergerea datelor dintr-o tabelă nu poate fi anulată


# Tranzacții – SQL Server

- Implicit – acest mod este activat cu comanda
  - ▣ SET IMPLICIT\_TRANSACTIONS ON
- Pentru a-l dezactiva
  - ▣ SET IMPLICIT\_TRANSACTIONS OFF
- Cu acest mod activat oricare din comenzile de mai jos va deschide în mod automat o tranzacție care rămâne deschisă până la executarea comenzilor COMMIT sau ROLLBACK




# Tranzacții – SQL Server

- Comenzi care generează tranzacții implicite
  - ALTER TABLE
  - FETCH
  - REVOKE
  - CREATE
  - GRANT
  - SELECT, DELETE, INSERT,
  - TRUNCATE TABLE
  - DROP, OPEN, UPDATE



# Tranzacții – SQL Server

- Explicit – sunt cele pe care le definește programatorul SQL
  - Acesta este modul recomandat pentru tranzacții
  - Putem controla cu exactitate care sunt instrucțiunile ce aparțin unei tranzacții și acțiunile pe care să le întreprindem dacă apar erori.
- 

# Tranzacții – SQL Server

- @@TRANCOUNT – numărul de tranzacții active pe o conexiune
- BEGIN TRANSACTION – incrementează valoarea
- ROLLBACK TRANSACTION și COMMIT TRANSACTION decrementează valoarea
- ROLLBACK TRAN la un savepoint nu are nici un efect asupra valorii

# Tranzacții – comenzi

- BEGIN TRANSACTION – stabilește punctul de pornire al unei tranzacții explicite
- ROLLBACK TRANSACTION – Readuce datele la starea inițială – de la începutul tranzacției. Orice modificări asupra datelor sunt anulate. Resursele alocate tranzacției sunt eliberate
- COMMIT TRANSACTION – termină tranzacția dacă nu a fost nici o eroare și modificările devin permanente. Resursele sunt eliberate

# Tranzacții – comenzi

- `SAVE TRANSACTION` – stabilește un punct de salvare într-o tranzacție. Se definește un punct la care se poate întoarce o tranzacție în cazul în care o parte a tranzacției este anulată. (Nu vor fi anulate toate comenzile din tranzacție)
- O tranzacție trebuie anulată sau executată imediat după o întoarcere la un `SAVEPOINT`

# Tranzacții-exemplu - tran1.sql

```
USE AdventureWorks
```

```
GO
```

```
-- Before count
```

```
SELECT COUNT(*) BeforeCount FROM HumanResources.Department
```

```
-- Variable to hold the latest error integer value
```

```
DECLARE @Error int
```

```
BEGIN TRANSACTION
```

```
INSERT HumanResources.Department (Name, GroupName)
```

```
VALUES ('Accounts Payable', 'Accounting')
```

```
SET @Error = @@ERROR
```

```
IF (@Error <> 0) GOTO Error_Handler
```

# Tranzacții – exemplu

```
INSERT HumanResources.Department (Name, GroupName)
```

```
VALUES ('Engineering', 'Research and Development')
```

```
SET @Error = @@ERROR
```

```
IF (@Error <> 0) GOTO Error_Handler
```

```
COMMIT TRAN
```

```
Error_Handler:
```

```
IF @Error <> 0
```

```
BEGIN
```

```
    ROLLBACK TRANSACTION
```

```
END
```

```
-- After count
```

```
SELECT COUNT(*) AfterCount FROM HumanResources.Department
```

# Tranzacții – funcția @@ERROR

- Funcția sistem @@ERROR întoarce valoarea erorii pentru ultima instrucțiune executată în contextul conexiunii curente
- Exemplu – nu se va insera nici o înregistrare




# Tranzacții – DBCC OPENTRAN

- Dacă o tranzacție rămâne deschisă poate bloca alte procese în a efectua activități asupra datelor modificate
- DBCC OPENTRAN – identifică cea mai veche tranzacție activă
- Exemplu:

```
BEGIN TRANSACTION  
DELETE Production.ProductProductPhoto  
WHERE ProductID = 317  
DBCC OPENTRAN('AdventureWorks')  
ROLLBACK TRAN
```



# Tranzacții - Locking

- Blocarea este o parte normală și necesară a unui SGBD relațional
  - Asigură integritatea datelor prin interzicerea actualizărilor concurente asupra aceluiași date și prin interzicerea vizualizării unor date ce sunt în curs de actualizare
- 

# Tranzacții - Locking

- Ajută la prevenirea apariției problemelor legate de concurență
  - un user încearcă să citească date modificate de altul
  - să modifice date pe care altul le citește
  - să modifice date pe care altul încearcă să le modifice

# Tranzacții - Locking

- Lock – sunt plasate asupra resurselor SQL Server
- Lock mode = Modul în care este blocată o resursă
  - Shared lock – plasat în timpul interogărilor read-only, care nu modifică datele. Alte procese pot citi dar nu pot actualiza datele

# Tranzacții - Locking

- Lock mode
  - Intent lock – se creează o coadă de lock-uri, ce desemnează ordinea conexiunilor și a drepturilor asociate de actualizare sau citire asupra resurselor. Se folosesc pentru a indica intenții viitoare de a bloca o anumită resursă
  - Update lock – sunt puse înainte de a face actualizări. Când se modifică o înregistrare se transformă în *exclusive lock*. Dacă nu se face modificare se trece la *shared lock*. Acest tip de lock previne deadlock-uri

# Tranzacții - Locking

- Lock mode
  - Exclusive lock – orice altă operație asupra resurselor este oprită. (la INSERT, UPDATE, DELETE)
  - Schema modification – când se execută operații DDL
  - Schema stability – la compilarea unei interogări. Nu se pot face operații DDL
  - Bulk update – la operații de copiere bulk. Crește performanța de copiere, scade concurența
  - Key-range – protejează un interval de înregistrări (pe baza unei chei de indexare)

# Tranzacții - Locking

- Lock se poate pune pe orice tip de resursă – de la o înregistrare până la tablă sau baza de date
- Orice lock necesită memorie
- Resurse asupra cărora se pun lock-uri
  - Allocation unit - o mulțime de pagini grupate după tip de date
  - Application – o resursă specificată de aplicație
  - DB – toată baza de date
  - Extent – o unitate de alocare de 8KB

# Tranzacții - Locking

- Resurse asupra cărora se pun lock-uri
  - File – fișierul bazei de date
  - HOBT – heap sau B-tree (o tabelă fără un index de tip clustered)
  - Metadata
  - Key – asupra unui index
  - Object – un obiect al bazei de date (view, procedură stocată, funcție)
  - Page – o pagină de date de 8KB sau o pagină de index
  - RID – row identifier
  - Table



# Tranzacții - Locking

- Nu toate tipurile de lock-uri sunt compatibile una cu alta.
- Pe o resursă care are un *exclusive lock*, nu se pot pune alte lock-uri
- Pe o resursă care are un *update lock* se poate pune doar un *shared lock* de către o altă tranzacție.
- Pe o resursă care are un *shared lock* se pot pune alte *shared* sau *update lock*.

# Tranzacții-Locking-Concurență

- Isolation – ACID
- Izolarea tranzacțiilor se referă la gradul la care modificările făcute de o tranzacție pot fi văzute de o altă tranzacție – în condiții de acces concurent la baza de date.
- Acest grad poate varia

# Tranzacții-Locking-Concurență

- ANSI/ISO SQL definește 4 tipuri de interacțiuni între tranzacții concurente
  - *Dirty reads* – o tranzacție modifică o valoare și o alta o citește înainte de commit. În cazul unui rollback în prima tranzacție se vor citi date care nu se salvează în BD
  - *Non-repeatable reads* – o tranzacție actualizează date și altă tranzacție citește datele. Datele citite înainte de actualizare și după actualizare vor fi diferite

# Tranzacții-Locking-Concurență

- ANSI/ISO SQL definește 4 tipuri de interacțiuni între tranzacții concurente
  - *Phantomreads* – o tranzacție citește de două ori. Între cele două operații de citire are loc o operație de actualizare (inserare/ștergere). Rezultatele celor două citiri vor fi diferite
  - *Lost updates* – două tranzacții actualizează o valoare. Ultima tranzacția care se execută "câștigă". Prima actualizare se pierde.

# Tranzacții-Locking-Concurență

- Pentru a evita aceste probleme de concurență SQL Server definește niveluri de izolare
  - READ COMMITTED (implicit) – se folosesc shared locks
  - READ UNCOMMITTED – cel mai puțin restrictiv. Oferă cel mai mare nivel de concurență și cel mai mic nivel de integritate a datelor. Util pentru date relativ statice – crește performanța
  - REPEATABLE READ – nu sunt permise dirty și non-repeatable reads

# Tranzacții-Locking-Concurență

- Niveluri de izolare
  - SERIALIZABLE - setarea cea mai restrictivă. Nu se permite inserări/modificări care ar afecta rezultatul unei interogări
  - SNAPSHOT – permite citirea unei versiuni a datelor așa cum au existat la începutul tranzacției

# Tranzacții-Locking-Concurență

```
SET TRANSACTION ISOLATION LEVEL
```

```
{
```

```
    READ UNCOMMITTED
```

```
    | READ COMMITTED
```

```
    | REPEATABLE READ
```

```
    | SNAPSHOT
```


```
    | SERIALIZABLE
```

```
}
```



# Tranzacții-Locking-Concurență


Exemplu – tran2.sql







# Tranzacții-Blocking

- Blocarea are loc atunci când o tranzacție pune lock pe resurse ce alte tranzacții vor să citească sau să modifice
  - Blocările pe termen scurt sunt de regulă OK și de așteptat pentru aplicații intensive
  - Aplicațiile proiectate defectuos pot cauza blocări de termen lung care păstrează lock-uri pe resurse și blochează alte sesiuni să citească sau să actualizeze date.
- 

# Tranzacții-Blocking

- Un proces blocat rămâne blocat nedefinit sau până când
  - expiră (pe baza SET LOCK\_TIMEOUT),
  - serverul este oprit,
  - procesul este oprit,
  - conexiunea termină actualizările

# Tranzacții-Blocking

- Motive pentru care poate apare blocare pe termen lung:
  - Lock-uri excesive pe liniile unei tabele fără index poate determina SQL Server să obțină un table lock, blocând alte tranzacții
  - Aplicațiile deschid o tranzacție după care așteaptă input de la user în timp de conexiunea rămâne deschisă
  - Aplicațiile folosesc tranzacții lungi care actualizează multe înregistrări sau multe tabele într-o tranzacție (împărțirea unei tranzacții mari în mai multe mici poate îmbunătăți concurența)

# Tranzacții-Blocking

- Identificarea problemelor de blocare
- `sys.dm_os_waiting_tasks` DMV pentru identificarea procesului ce a cauzat blocarea
- Pentru identificarea textului comenzii care au cauzat blocarea
  - `sys.dm_exec_sql_text`
  - `sys.dm_exec_connections`

# Tranzacții-Blocking

- KILL {spid | UOW} [WITH STATUSONLY]
- Oprește o sesiune activă
- Se folosește doar când alte metode nu dau rezultate
  - ▣ spid id-ul de sesiune asociat cu conexiunea activă ce trebuie oprită
- Exemplu: tran3.sql

# Tranzacții-Blocking

- Când o tranzacție/instrucțiune este blocată așteaptă ca un lock pe o resursă să fie eliberat
- SET LOCK\_TIMEOUT specifică timpul de așteptare înainte de a returna eroare
- Sintaxa:
  - ▣ SET LOCK\_TIMEOUT timeout\_period (în ms)
- Exemplu: tran4.sql

# Tranzacții-Deadlock

- Are loc atunci când
  - Sesiunea 1 are un lock pe resurse ce trebuie modificate de Sesiunea 2
  - Sesiunea 2 are un lock pe resurse ce trebuie modificate de Sesiunea 1
- Niciuna din cele două sesiuni nu poate continua
- Una din sesiuni va fi aleasă ca *deadlock victim*
- Sesiunea va fi oprită și tranzacția anulată

# Tranzacții-Deadlock

- Motive pentru care apare deadlock
  - Aplicația accesează tabele în ordine diferită – de ex. Sesiunea 1 actualizează Customers și apoi Orders, iar Sesiunea 2 actualizează Orders și apoi Customers
  - Aplicația folosește tranzacții lungi ce actualizează mai multe linii sau mai multe tabele



# Tranzacții-Deadlock

- Identificare deadlock
  - DBCC TRACEON
  - DBCC TRACEOFF
  - DBCC TRACESTATUS
- DBCC TRACEON ( trace# [ ,...n ] [ , -1 ] )  
[ WITH NO\_INFOMSGS ]
- DBCC TRACESTATUS ( [ [ trace# [ ,...n ] ] [ , ] [ -1 ] ] )  
[ WITH NO\_INFOMSGS ]

# Tranzacții-Deadlock

- DBCCTRACEOFF ( trace# [ ,...n ] [ , -1 ] )  
[ WITH NO\_INFOMSGS ]
- trace# - indică una sau mai multe numere *trace flag* ce trebuie activate/dezactivate/verificate
- -1 activare/dezactivare/verificare globală
- Exemplu: tran5.sql

# Tranzacții-Deadlock

- Putem mări șansa ca o sesiune să fie aleasă ca victimă pentru deadlock
- SET DEADLOCK\_PRIORITY  
{ LOW | NORMAL | HIGH | <numeric-priority> }
- Va fi aleasă ca victimă sesiunea cu valoarea cea mai mică
- *numeric-priority* – valori de la -10 la 10

# Proceduri stocate

- O PS grupează una sau mai multe comenzi SQL într-o unitate logică, stocată ca un obiect al bazei de date
- Definiția este accesibilă din `sys.sql_modules`
- La prima execuție se creează un plan de execuție stocat în memoria cache de plan
- Planul poate fi refolosit la execuții ulterioare
- Reutilizarea planului permite performanțe mai bune în comparație cu interogările ad-hoc necompileate

# Proceduri stocate


- Avantaje

- Ajută la centralizarea codului SQL în data tier. Aplicațiile ce includ cod SQL ad-hoc sunt greu de depanat și întreținut.
- Reduc traficul pe rețea pentru interogări ad-hoc formate din multe linii. (E mai eficient să se trimită de la aplicație o singură linie pentru executarea unui proceduri stocate decât 500 de linii de cod SQL).



# Proceduri stocate

- Avantaje


- Încurajează reutilizarea codului
  - Permit obscurarea metodei de obținere a datelor. Permite modificări la nivelul datelor fără a fi nevoie de modificarea codului la nivel de aplicație
  - Pot utiliza controlul accesului, tabele temporare, variabile de tip tabel etc.
  - Securitate sporită. Pot funcționa ca un nivel de control la date/tabele
- 

# Proceduri stocate

- Sintaxa pentru creare  
CREATE PROCEDURE [schema\_name.]  
    procedure\_name  
    AS { <sql\_statement> [ ...n ] }
- Sintaxa pentru executare  
EXEC [schema\_name.] procedure\_name
- Exemplu: StoredProc1.sql



# Proceduri stocate

- SET NOCOUNT ON
  - Oprește afișarea/transmiterea mesajelor în legătură cu numărul de linii afectate de ultima comandă SQL
  - Funcția @@ROWCOUNT se va actualiza indiferent de setarea pentru NOCOUNT
  - Traficul prin rețea va fi redus – aplicația va fi mai rapidă
- 



# Proceduri stocate

- Proceduri stocate parametrizate
- Sintaxa:

```
CREATE { PROC | PROCEDURE } [schema_name.]  
    procedure_name [ ; number ]  
    [ { @parameter [ type_schema_name. ] data_type }  
    [ VARYING ] [ = default ] [ OUT | OUTPUT ] [ READONLY ]  
    ] [ , ...n ]  
    [ WITH <procedure_option> [ , ...n ] ]  
    [ FOR REPLICATION ]  
    AS { <sql_statement> [ ; ] [ ...n ] | <method_specifier> }
```

# Proceduri stocate

- Proceduri stocate parametrizate
- Parametri sunt prefixați de caracterul @
- Parametri sunt de două tipuri
  - Input – de intrare
  - Output – de ieșire
- Parametrul poate specifica o valoare implicită caz în care la apel nu mai trebuie inclus
- Exemplu: StoredProc2.sql

# Proceduri stocate


- Parametri de ieșire – OUTPUT | OUT
- Permit transferul de informații în rutina apelantă
- Exemplu: StoredProc3.sql
- Dintr-o procedură stocată se pot transmite mai multe result-set-uri care pot fi consumate de aplicație

# Proceduri stocate

- Modificarea unei proceduri stocate
  - ALTER PROCEDURE
- Putem schimba orice în afară de nume
- Eliminarea unei proceduri stocate
  - DROP PROCEDURE { [ schema\_name. ]  
procedure } [ ,...n ]
- Exemplu: StoredProc4.sql



# Proceduri stocate

- Executarea automată a unei proceduri stocate la pornirea SQL Server
  - Se face cu procedura stocată sistem `sp_procoption`
  - Pot rula doar în baza de date master
  - Exemplu: `StoredProc5.sql`
- 

# Proceduri stocate

- Metadata despre proceduri stocate  
SELECT definition, execute\_as\_principal\_id,  
is\_recompiled, uses\_ansi\_nulls,  
uses\_quoted\_identifier  
FROM sys.sql\_modules m  
INNER JOIN sys.objects o ON  
m.object\_id = o.object\_id  
WHERE o.type = 'P'

# Proceduri stocate

- Documentare (recomandare)

```
CREATE PROCEDURE dbo.usp_IMP_DWP_FactOrder  
AS
```

```
-----
```

```
-- Purpose: Populates the data warehouse, Called by Job
```

```
--
```

```
-- Maintenance Log
```

```
--
```

```
-- Update By      Update Date      Description
```

```
-- -----
```

```
-- -----
```

```
-- -----
```


```
-- Joe Sack      8/15/2008      Created
```

```
-- Joe Sack      8/16/2008      A new column was added to  
--the base table, so it was added here as well.
```

```
... Transact-SQL code here
```



# Proceduri stocate

- Securitate – PS au avantaje inerente în ceea ce privește securitatea
  - Codul inline ad-hoc este mult mai afectat de atacuri de tip 'SQL injection'
  - Includerea codului SQL în PS permite ascunderea lui de influențe externe
  - Putem controla modul în care sunt făcute modificările și modul în care sunt extrase datele.
- 




# Proceduri stocate

- “Criptarea” procedurilor stocate
- Prin criptarea definiției nu se va putea accesa codul procedurii stocate
- Se previne modificarea PS și operațiile de tip reverse-engineering.
- Sintaxa:  

```
CREATE PROCEDURE proc_name  
WITH ENCRYPTION  
AS ...
```
- Exemplu: StoredProc7.sql



# Proceduri stocate

- Clauza EXECUTE AS – stabilirea contextului de securitate pentru PS
  - Context de securitate – permisiunile user-ului ce execută procedura stocată
  - O PS poate fi executată
    - În contextul de securitate al apelantului
    - User-ul care a creat/modificat procedura
    - Un anumit login
    - Proprietarul PS
- 

# Proceduri stocate

- Exemplu: StoredProc8.sql
- SQL dinamic – se va asigura faptul că apelantul are dreptul EXECUTE pe PS și drepturile necesare pentru a executa operațiile din secvența de cod SQL dinamic
- Exemplu: StoredProc9.sql
- Un user explicit poate fi desemnat ca și context de securitate pentru PS
- Exemplu: StoredProc9.sql

# Proceduri stocate

- EXECUTE AS { CALLER | SELF | OWNER | 'user\_name' }
- Implicit este CALLER – se folosesc permisiunile user-ului care apeleaza procedura stocată. Dacă userul nu are drepturile necesare execuția va eșua
- SELF - contextul de execuție va fi al user-ului care a creat/modificat PS
- OWNER – proprietarul schemei PS

# Funcții definite de utilizator

- Există trei tipuri de funcții
  - Scalare
  - Inline table-valued
  - Multi-statement table-valued
- Funcțiile scalare întorc o singură valoare pe baza a zero, unu sau mai mulți parametri
- Inline table-valued – întorc un tip de date tabel pe baza unei singure comenzi SELECT care definește liniile și coloanele

# Funcții definite de utilizator

- O funcție inline poate fi folosită în clauza FROM (spre deosebire de o PS)
- Poate fi folosită într-un join
- Poate primi parametri (spre deosebire de un view)
- Multi-statement UDF – întorc un rezultat de tip tabelar (result-set). Permit mai multe comenzi/instrucțiuni care vor construi rezultatul

# Funcții definite de utilizator

- Funcțiile pot fi folosite acolo unde PS nu pot fi folosite. De ex. clauza SELECT sau FROM
- Încurajează reutilizarea codului
- Funcții scalare
  - Primesc zero, unu sau mai mulți parametri
  - Folosite de regulă pentru a converti/translata o valoare într-alta
  - Pot fi folosite în expresii de căutare, coloane, join, constrângeri de verificare sau de valoare implicită

# Funcții definite de utilizator

- Sintaxa simplificată pentru funcții scalare

```
CREATE FUNCTION [ schema_name. ] function_name  
( [ { @parameter_name [ AS ] [ type_schema_name. ]  
      parameter_data_type  
      [ = default ] [ READONLY ] }  
      [ ,...n ] ] )  
RETURNS return_data_type  
[ WITH <function_option> [ ,...n ] ]  
[ AS ]  
BEGIN  
    function_body  
    RETURN scalar_expression  
END
```



# Funcții definite de utilizator

- Exemplu: UDF1.sql – verifică dacă o comandă SQL este suspectă după anumite criterii
- Exemplu: UDF2.sql – transformă un string astfel încât fiecare cuvânt începe cu literă mare
- O funcție scalară poate fi folosită
  - Expresie coloană în SELECT sau GROUP BY
  - Condiție în JOIN, FROM
  - Condiție în WHERE, HAVING

# Funcții definite de utilizator

- O funcție inline întoarce un tip de date tabel
- Tabela nu se definește explicit
- Liniile și coloanele sunt definite de o singură comandă SELECT
- Poate primi parametri
- Sunt foarte similare cu vizualizările
- Sunt referite în clauza FROM

# Funcții definite de utilizator


- Sintaxa simplificată

```
CREATE FUNCTION [ schema_name. ] function_name  
( [ { @parameter_name [ AS ]  
[ type_schema_name. ] scalar_parameter_data_type [ =  
    default ]  
} [ ,...n ] ] )  
RETURNS TABLE  
[ AS ]  
RETURN [ ( ) select_stmt [ ) ]
```

- Exemplu: UDF3.sql



# Funcții definite de utilizator

- Funcții multi-instrucțiune
  - Pot fi folosite în clauza FROM
  - Pot folosi mai multe instrucțiuni pentru a genera un result-set
- 

# Funcții definite de utilizator

- Sintaxa

```
CREATE FUNCTION [ schema_name. ] function_name  
( [ { @parameter_name [ AS ] [ type_schema_name. ]  
    parameter_data_type  
    [ = default ] [ READONLY ] } [ ,...n ] ] )  
RETURNS @return_variable TABLE  
    <table_type_definition>  
[ WITH <function_option> [ ,...n ] ]  
[ AS ]  
BEGIN  
function_body  
RETURN  
END
```

# Funcții definite de utilizator

- Exemplu: UDF4.sql
- Funcția primește două argumente
  - Un string
  - Un caracter care este delimitatorul
  - Funcția întoarce un result-set format din substring-uri
- Funcțiile multi-instrucțiune permit crearea de secvențe de cod mult mai sofisticate

# Funcții definite de utilizator

- Modificarea unei funcții: UDF5.sql
- Nu se poate schimba numele
- Nu se poate transforma o funcție scalară într-una de tip table-valued sau invers
- Metadate pentru funcții: UDF6.sql
- Eliminarea unei funcții
  - `DROP FUNCTION { [ schema_name. ]  
function_name } [ ,...n ]`


# Funcții definite de utilizator

- Exemple de utilizare a funcțiilor
  - Reducerea repetării de cod: UDF7.sql
  - Chei surogat – chei naturale: UDF8.sql
  - Înlocuirea unor vizualizări cu funcții multi-instrucțiune: UDF9.sql





# Tipuri definite de utilizator

- Utile pentru definirea în mod consistent a tipurilor de date
  - De ex: PIN, PhoneNumber, EmailAddress
  - O data creat un tip de date poate fi folosit pentru coloane ale tabelelor, parametri, definiții de variabile
  - Se creează un alias pentru care se specifică tipul de date, lungimea, atributul NULL
- 

# Tipuri definite de utilizator

- Sintaxa:

```
CREATE TYPE [ schema_name. ] type_name
{
    FROM base_type
    [ (precision [ ,scale ] ) ]
    [ NULL | NOT NULL ]
}
```

Exemplu: UDT1.sql

# Tipuri definite de utilizator

- Identificarea obiectelor din baza de date care folosesc un tip de date definit de utilizator

```
SELECT OBJECT_NAME(c.object_id) Table_Name,  
       c.name Column_Name
```

```
FROM sys.columns c
```

```
INNER JOIN sys.types t ON
```

```
c.user_type_id = t.user_type_id
```

```
WHERE t.name = 'AccountNBR'
```

Exemplu: UDT2.sql

# Tipuri definite de utilizator

- Identificarea parametrilor de proceduri și funcții de un anumit tip

```
SELECT OBJECT_NAME(p.object_id) Table_Name,  
       p.name Parameter_Name  
FROM sys.parameters p  
INNER JOIN sys.types t ON  
p.user_type_id = t.user_type_id  
WHERE t.name = 'AccountNBR'
```

# Tipuri definite de utilizator

- `sys.columns` conține o linie pentru fiecare coloană definită pentru
  - Funcție de tip `table-valued`
  - Tabelă
  - Vizualizare
- `sys.types` conține o linie pentru fiecare
  - Tip de date definit de utilizator
  - Tip de date sistem
- `sys.parameters` conține o linie pentru fiecare obiect al bazei de date ce acceptă parametri

# Tipuri definite de utilizator

- Eliminarea unui UDT
  - `DROP TYPE [ schema_name. ] type_name`
- Înainte de a elimina un tip de date trebuie eliminate toate referințele din baza de date la acel tip
- Exemplu: UDT3.sql

# Tipuri definite de utilizator

- Transmiterea unor parametri de tip tabel
- Putem transmite procedurilor stocate și funcțiilor parametri de tip tabel
- Parametrul de tip tabel trebuie calificat cu atributul READONLY pentru ca nu putem modifica o astfel de valoare în PS sau funcție
- Exemplu: UDT4.sql




# Trigger

- DML trigger – conțin cod SQL ce se execută în cazul operațiilor INSERT, UPDATE sau DELETE asupra unei tabele sau vizualizări
- DDL trigger – răspund la evenimente ale serverului sau ale bazei de date
- Pot răspunde automat la anumite acțiuni
- Utile atunci când trebuie să asigurăm anumite reguli de business





# Trigger

- Trigger DML
  - Atunci când are loc o operație de modificare a datelor triggerul va efectua operațiile definite
  - Trigger-ul se definește în cod SQL și poate efectua o serie de operații la fel ca și PS
  - Definit pentru una sau mai multe operații
    - FOR UPDATE
    - FOR INSERT
    - FOR DELETE
- 

# Trigger

- Există două tipuri de triggere DML
  - AFTER - doar pentru tabele
  - INSTEAD OF – pentru tabele și view
- Exemplu:
  - Trigger care adaugă informații într-o tabelă de jurnalizare la operații DML
  - Trigger care decrementează valoarea unei cantități

# Trigger

- Aspecte de care trebuie ținut cont
  - Adesea pot deveni o problemă ascunsă și prin urmare uitată
  - Dacă orice modificare a datelor se face cu PS este recomandat să nu se folosească trigger. Astfel este mai simplu de depanat/întreținut aplicația
  - Performanța trebuie să fie primordială. Triggere ce durează mult pot încetini operațiile DML
  - Operațiile fără log WRITTEXT, TRUNCATE, bulk insert nu lansează trigger



# Trigger

- Aspecte de care trebuie ținut cont
  - Constrângerile sunt mai rapide decât triggerele. O regulă de business este mai bine să fie asigurată cu o constrângere decât cu un trigger
  - Result-set-ul unui SELECT nu ar trebui să fie returnat de trigger. Aplicațiile nu pot consuma aceste result-set-uri într-un mod elegant
- Triggerele sunt un mod foarte bun de asigurare a regulilor de business în mod programatic

# Trigger

- Trigger AFTER se execută după rularea cu succes a unei comenzi INSERT, UPDATE sau DELETE
- Sintaxa:

```
CREATE TRIGGER [ schema_name . ]trigger_name  
ON table  
[ WITH <dml_trigger_option> [ ...,n ] ]  
AFTER  
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }  
[ NOT FOR REPLICATION ]  
AS { sql_statement [ ...n ] }
```

# Trigger

- Opțiuni pentru trigger:
  - ENCRYPTION
  - EXECUTE AS
- SQL Server creează două tabele virtuale ce sunt disponibile în trigger: inserted și deleted

Operația DML	inserted conține	deleted conține
INSERT	Liniile inserate	---
UPDATE	Liniile noi	Liniile vechi
DELETE	---	Liniile șterse

# Trigger

- Un trigger trebuie să țină cont de faptul că o operație DML poate afecta zero, una sau mai multe înregistrări
- Exemplu: trigger1.sql
- Trigger INSTEAD OF
  - se utilizează în locul operației DML care a lansat trigger-ul
  - Poate fi creat pentru tabele sau vizualizări

# Trigger

- Sintaxa pentru trigger INSTEAD OF  
CREATE TRIGGER [ schema\_name . ]trigger\_name  
ON { table | view }  
[ WITH <dml\_trigger\_option> [ ...,n ] ]  
INSTEAD OF  
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }  
[ NOT FOR REPLICATION ]  
AS { sql\_statement [ ...n ] }
- Exemplu: trigger2.sql



# Trigger

- Tranzacții și triggere
- Când este lansat un trigger SQL Server creează o tranzacție ceea ce permite anularea operațiilor efectuate de trigger sau de apelant
- Folosim ROLLBACK pentru a anula trigger-ul și operațiile efectuate de acesta
- Dacă operația care a lansat trigger-ul este într-o tranzacție explicită aceasta va fi anulată în cazul în care trigger-ul face ROLLBACK
- Exemplu: trigger3.sql

# Trigger

- Controlul triggerelor DML pe baza coloanelor modificate
- Funcția UPDATE ne permite să determinăm dacă o anumită coloană a fost inserată sau actualizată
- Exemplu: trigger4.sql
- Metadata pentru trigger
  - Vizualizarea sys.triggers
  - Exemplu: trigger5.sql

# Trigger

- DDL trigger – răspund la evenimente ale serverului sau ale bazei de date
- Pot răspunde la comenzi
  - La nivel de bază de date: CREATE TABLE sau DROP TABLE (stocate în baza de date pentru care sunt create)
  - La nivel de server: crearea unui nou login (stocate în baza de date master)

# Trigger

- Sintaxa pentru trigger DDL  
CREATE TRIGGER trigger\_name  
ON { ALL SERVER | DATABASE }  
[ WITH <ddl\_trigger\_option> [ ...,n ] ]  
FOR { event\_type | event\_group } [ ,...n ]  
AS { sql\_statement [ ...n ] }
- Opțiuni: ENCRYPTION, EXECUTE AS

# Trigger

- event\_type un eveniment la care va reacționa triggerul
  - CREATE\_TABLE, ALTER\_TABLE, DROP\_INDEX
- event\_group – grupare logică de mai multe event\_type
  - DDL\_PARTITION\_FUNCTION\_EVENTS
    - CREATE\_PARTITION\_FUNCTION
    - ALTER\_PARTITION\_FUNCTION
    - DROP\_PARTITION\_FUNCTION
- Lista completă: SQL Server Books Online
  - DDL Events for Use with DDL Triggers
  - Event Groups for Use with DDL Triggers

# Trigger

- Trigger DDL pentru auditarea evenimentelor la nivelul bazei de date
  - ▣ Exemplu: trigger6.sql
- Trigger DDL pentru auditarea de evenimente la nivel de server
  - ▣ Exemplu: trigger7.sql
- Logon trigger
  - ▣ Exemplu: trigger8.sql
- Metadata trigger DDL: trigger9.sql

# Trigger

- Modificare trigger
  - ALTER TRIGGER
  - Exemplu: trigger10.sql
- Activare/dezactivare trigger
  - Uneori triggerurile trebuie dezactivate din diverse motive
    - Depanare
    - Import de date care nu ar trebui să lanseze triggerul

# Trigger

- Sintaxa activare/dezactivare trigger

DISABLE TRIGGER

[schema.]trigger\_name

ON { object\_name | DATABASE | SERVER }

ENABLE TRIGGER

[schema\_name.]trigger\_name

ON { object\_name | DATABASE | SERVER }

Exemplu: trigger11.sql




# Trigger

- Încuibarea triggerelor
  - Operațiile dintr-un trigger pot lansa alte trigger
  - Nivelul maxim de încuibare este 32
  - Se poate dezactiva total încuibarea
  - Exemplu: trigger12.sql
- Trigger recursiv
  - Încuibarea este considerată recursivă atunci când un trigger se reapelează pe el însuși
  - Direct sau indirect
  - SET RECURSIVE\_TRIGGERS ON
  - Exemplu: trigger13.sql



# Trigger

- Ordinea de lansare a triggerelor
  - De regulă pentru un eveniment ar trebui să existe un singur trigger
  - Dar, putem defini și mai multe triggere pentru același eveniment
  - În acest caz ne interesează și ordinea în care sunt rulate
- 

# Trigger

- Ordinea de rulare a triggerelor

```
sp_settriggerorder [ @triggername = ] '[ triggerschema.  
    ] triggername'
```

```
, [ @order = ] 'value'
```

```
, [ @stmttype = ] 'statement_type'
```

```
[ , [ @namespace = ] { 'DATABASE' | 'SERVER' | NULL } ]
```

- ▣ value = {first, last, none}
- ▣ statement\_type = INSERT, UPDATE, DELETE, CREATE\_INDEX, ALTER\_INDEX etc.
- ▣ Orice trigger în afară de primul și ultimul va fi rulat într-o ordine aleatoare


- Exemplu: trigger14.sql

# Trigger

- Eliminarea unui trigger
- Trigger DML
  - DROP TRIGGER  
schema\_name.trigger\_name [ ,...n ]
- Trigger DDL
  - DROP TRIGGER trigger\_name [ ,...n ]  
ON { DATABASE | ALL SERVER }
- Exemplu: trigger14.sql



# Tratarea erorilor

- Mesaje de eroare sistem și definite de user
  - `sys.messages` conține o linie pentru fiecare mesaj de eroare
  - Mesajele de eroare sistem (built-in) sunt lansate ca răspuns la erori standard
  - Mesajele de eroare definite de user sunt definite în diverse aplicații – permit parametrizare
  - Exemplu: `error1.sql`
- 

# Tratarea erorilor

- Nivelul de severitate al erorii: de la 0 la 25
  - 0-10 – mesaje informative
  - 11-16 – erori ale motorului bazei de date ce pot fi corectate de user (obiecte ale bazei de date care lipsesc când se execută un query, deadlock, erori de sintaxă).
    - Violarea unei chei primare nivel 14
    - Împărțire la zero nivel 16
  - 17-19 – necesită atenția sysadmin-ului
    - Nu mai sunt resurse de memorie
    - Au fost atinse diverse limite ale serverului

# Tratarea erorilor

- Nivelul de severitate al erorii: de la 0 la 25
  - 20-25 – erori fatale și probleme de sistem
    - Probleme hardware sau software
    - Probleme de integritate
    - Discuri defecte
- Mesajele de eroare pot conține caracterul % - indică faptul că e vorba de un parametru ce va fi substituit (%s string, %d sau %i întreg cu semn, %o octal, %u unsigned, %x hexa)

# Tratarea erorilor

- Crearea unui mesaj de eroare definit de user  
sp\_addmessage [ @msgnum = ] msg\_id ,  
[ @severity = ] severity ,  
[ @msgtext = ] 'msg'  
[ , [ @lang = ] 'language' ]  
[ , [ @with\_log = ] 'with\_log' ]  
[ , [ @replace = ] 'replace' ]
- Mesajul de eroare poate fi invocat cu  
RAISERROR



# Tratarea erorilor

- msg\_id poate fi între 50.001 și 2.147.483.647
- with\_log – indică dacă mesajul va fi scris în Windows Application Error log
- replace – se face o înlocuire a unui mesaj existent pe baza id și lang
- Exemplu: error2.sql
- RAISEERROR folosit pentru a lansa erori legate de aplicație (business logic)

# Tratarea erorilor

- Eliminarea unui mesaj de eroare  
`sp_dropmessage [ @msgnum = ] message_number`  
`[ , [ @lang = ] 'language' ]`
  - ▣ language poate avea valoarea 'ALL' caz în care se elimină mesajul pentru toate limbile
- Exemplu: `EXEC sp_dropmessage 100001`

# Tratarea erorilor

- Lansarea manuală a erorilor – RAISERROR
- Putem invoca un mesaj de eroare definit de user din sys.messages sau un mesaj de eroare dintr-un string

- Sintaxa

```
RAISERROR ( { msg_id | msg_str | @local_variable }  
           { ,severity ,state }  
           [ ,argument [ ,...n ] ] )  
           [ WITH option [ ,...n ] ]
```

# Tratarea erorilor

- state – o valoare de la 1 la 127 ce identifică partea din cod de unde a fost lansată eroarea – atunci când codul este împărțit în mai multe secțiuni
- argument – parametri de substituție pentru mesaj
- WITH option
  - LOG – se scrie în SQL Server log și Windows log
  - NOWAIT – mesajul este trimis imediat la client
  - SETERROR – este setată variabila @@ERROR și ERROR\_NUMBER

# Tratarea erorilor

- Invocarea unui mesaj de eroare
  - Exemplu: error3.sql
- Interceptarea și tratarea erorilor de aplicație
  - TRY-CATCH poate fi folosit pentru a intercepta erorile de execuție din codul SQL
  - Se poate intercepta orice eroare cu severitate mai mare de 10
  - Cu excepția situației în care eroarea termină sesiunea utilizator
  - Pot fi interceptate și erorile invocate cu RAISERROR

# Tratarea erorilor

- Sintaxa

```
BEGIN TRY
```

```
{ sql_statement | statement_block }
```

```
END TRY
```

```
BEGIN CATCH
```

```
{ sql_statement | statement_block }
```

```
END CATCH
```

- În blocul CATCH putem raporta sau jurnaliza eroarea. Putem face ROLLBACK pe o tranzacție deschisă.

# Tratarea erorilor

- TRY...CATCH permite folosirea funcțiilor pentru erori și pentru starea tranzacțiilor
  - ERROR\_LINE
  - ERROR\_MESSAGE
  - ERROR\_NUMBER
  - ERROR\_PROCEDURE
  - ERROR\_SEVERITY
  - ERROR\_STATE
  - XACT\_ABORT – starea tranzacțiilor deschise din blocul TRY

# Tratarea erorilor

- Tratarea a erorilor învechită:
  - Se bazează în general pe funcția @ERROR a cărei valoare se testează după fiecare operație și pe instrucțiunea GOTO
  - Exemplu:error4.sql





# Tratarea erorilor

- Tratarea erorilor cu TRY...CATCH
  - Exemplu: error5.sql (aceeași funcționalitate ca și în error4.sql).
  - Partea de tratare a erorilor nu mai este amestecată cu logica aplicației

# Tratarea erorilor

- Apelul unei proceduri stocate poate fi pus într-un bloc try...catch – în felul acesta codul PS nu mai trebuie modificat pentru a include tratare a erorilor: error6.sql
- Blocurile try...catch pot fi include una într-alta: error7.sql

# Procesare condiționată. Controlul execuției.

- Procesare condiționată
  - Instrucțiunea CASE
  - Instrucțiunea IF...ELSE
  - Funcția CASE
- Controlul execuției
  - RETURN
  - WHILE
  - WAITFOR
  - GOTO

# Procesare condiționată

- Comanda CASE – returnează o valoare pe baza valorii uneia sau mai multor expresii
- Sintaxa

```
CASE input_expression  
WHEN when_expression THEN result_expression  
[ ...n ]  
[  
ELSE else_result_expression  
]  
END
```

# Procesare condiționată

- Exemplu: CondProc1.sql

```
SELECT DepartmentID, Name, GroupName,  
       CASE GroupName  
         WHEN 'Research and Development' THEN 'Room A'  
         WHEN 'Sales and Marketing' THEN 'Room B'  
         WHEN 'Manufacturing' THEN 'Room C'  
         ELSE 'Room D'  
       END ConferenceRoom  
FROM HumanResources.Department
```

# Procesare condiționată

- CASE pentru evaluarea unor expresii booleene

- Sintaxa:

CASE

WHEN Boolean\_expression THEN result\_expression

[ ...n ]

[

ELSE else\_result\_expression

]

END

# Procesare condiționată

- Dacă nu există clauza ELSE și toate valorile booleene sunt FALSE se va returna NULL
- Dacă mai multe expresii se evaluează la TRUE rezultatul va fi dat de prima expresie cu valoarea TRUE
- Exemplu: CondProc2.sql

# Procesare condiționată

- IF...ELSE

- Sintaxa:

IF Boolean\_expression

{ sql\_statement | statement\_block }

[ ELSE

{ sql\_statement | statement\_block } ]

- Exemplu: CondProc3.sql



# Controlul execuției

- RETURN – iese imediat din batch-ul curent sau din procedura stocată fără a mai executa instrucțiunile care urmează
- Exemplu: CondProc4.sql

```
IF NOT EXISTS
(SELECT ProductID
FROM Production.Product WHERE Color = 'Pink')
BEGIN
    RETURN
END
-- Nu se execută
SELECT ProductID FROM Production.Product WHERE Color = 'Pink'
```

# Controlul execuției

- RETURN [ integer\_expression ]
- Expresia întreagă poate fi folosită într-o procedură stocată pentru a transmite informații rutinei apelante
- Exemplu: CondProc5.sql

-- procedură stocată temporară ce lansează o eroare logică

```
CREATE PROCEDURE #usp_TempProc AS
```

```
    SELECT 1/0
```

```
    RETURN @@ERROR
```

```
GO
```

```
DECLARE @ErrorCode int
```

```
EXEC @ErrorCode = #usp_TempProc
```

```
PRINT @ErrorCode
```

# Controlul execuției

- WHILE – permite repetarea unor operații atâta timp cât o condiție este adevărată
- Sintaxa:

WHILE Boolean\_expression

{ sql\_statement | statement\_block }

[ BREAK ]


{ sql\_statement | statement\_block }

[ CONTINUE ]

{ sql\_statement | statement\_block }



# Controlul execuției

- BREAK – permite ieșirea forțată din instrucțiunea WHILE
  - CONTINUE – permite repornirea buclei WHILE
  - Exemplu: CondProc6.sql
  - Exemplu: CondProc7.sql
- 

# Controlul execuției

- GOTO – sare la o etichetă din codul TSQL
- Sintaxa:  
GOTO label  
label definition: code
- Exemplu: CondProc8.sql
- Recomandare: dacă există alternativă la GOTO ar trebui folosită
- GOTO scade gradul de claritate al codului

# Controlul execuției

- WAITFOR – întârzie executarea unor instrucțiuni pentru o anumită perioadă de timp sau până la o anumită oră

- Sintaxa:

```
WAITFOR
```

```
{
```

```
  DELAY 'time_to_pass'
```

```
  | TIME 'time_to_execute'
```

```
  | ( receive_statement ) [ , TIMEOUT timeout ] -- Service Broker
```

```
}
```

# Controlul execuției

- Exemple:

```
WAITFOR DELAY '00:00:10'
```

```
BEGIN
```

```
    SELECT TransactionID, Quantity
```

```
    FROM Production.TransactionHistory
```

```
END
```

---

```
WAITFOR TIME '19:01:00'
```

```
BEGIN
```


```
    SELECT COUNT(*)
```

```
    FROM Production.TransactionHistory
```

```
END
```




# Indexare

- Indecșii ajută la procesarea query-urilor prin creșterea vitezei de acces la datele stocate în tabele și vizualizări
  - Indecșii permit accesul ordonat la date pe baza unei ordini a datelor din linii
  - Liniile sunt ordonate pe baza valorilor stocate în anumite coloane
  - Coloanele reprezintă coloanele cheie ale indexului
- 





# Indexare


- Indexul este un obiect al bazei de date
  - Oferă acces mai rapid la date și pot eficientiza execuția interogărilor
  - Îi folosim pentru a oferi metode mai eficiente de acces la date
  - Liniile de care avem nevoie pot fi accesate mai rapid fără a fi nevoie de a accesa toate paginile de date ale unei tabele
- 

# Indexare

- În mod implicit liniile unei tabele neindexate nu sunt stocate într-o anumită ordine
- O tabelă pe care nu este o ordine se numește *heap*
- Pentru a extrage date dintr-un heap pe baza unei condiții de căutare trebuie parcurs tot tabelul – chiar și atunci când doar o singură înregistrare îndeplinește condiția
- O astfel de căutare se numește – *full table scan*



# Indexare


- Dacă avem un index pe coloana ce reprezintă condiția de căutare atunci se pot găsi înregistrările mult mai eficient
  - În MSSQL o tabelă face parte din una sau mai multe partiții
  - Partiție = o unitate de organizare ce permite separarea orizontală a alocării de date dintr-un tabel sau index, păstrând un singur obiect logic
- 

# Indexare

- La crearea unui tabel datele sunt păstrate în mod implicit într-o singură partiție
- O partiție conține heap-uri sau structuri de tip *B-tree* – atunci când sunt creați indecși
- La crearea unui index datele cheii de indexare sunt stocate într-o structură de tip B-tree
- O structură de tip B-tree are un nod rădăcină, care e punctul de pornire al indexului




# Indexare

- Nodul rădăcină are date index ce conțin un interval de valori de chei de indexare ce pointează la următorul nivel de noduri index numit nivel frunză intermediar
  - Nivelul cel mai de jos este nivelul frunză (nivelul nodurilor terminale)
  - Pentru un index de tip *clustered* nivelul nodurilor terminale este reprezentat de paginile de date ale tablei
- 



# Indexare


- Într-un index de tip *non-clustered*, nivelul nodurilor terminale conține pointeri la heap sau la paginile de date ale indexului clustered
  - Un index de tip clustered determină modul în care sunt stocate fizic efectiv datele tabeli
  - Putem avea un singur index clustered/tabelă
  - Acest tip de index stochează datele conform coloanei/coloanelor ce reprezintă cheia de indexare
- 

# Indexare

- Alegerea indexului clustered pentru o tabelă este una critică deoarece putem avea un singur index clustered
- Candidați pentru indexul clustered
  - Coloanele ce sunt folosite în interogări pe intervale (folosesc operatorii BETWEEN, <, >)
  - Coloane folosite pentru ordonarea unor result-seturi mari
  - Coloane folosite în funcții de agregare
  - Coloane ce conțin valori unice




# Indexare

- Coloane ce nu sunt bune pentru un index clustered
    - Coloane ce se actualizează frecvent
    - Coloane cu valori duplicate
  - Ar trebui să evităm să creăm indecși clustered cu prea multe coloane
- 





# Indexare


- Indecșii nonclustered stochează paginile de index separat față de datele fizice
  - Paginile și nodurile indexului conțin pointeri la datele fizice
  - Într-un index non-clustered nivelul nodurilor terminale este reprezentat de cheia de indexare + un locator de linii ce pointează la o linie
- 

# Indexare

- Candidați pentru index non-clustered
  - Coloane folosite frecvente în clauzele WHERE, JOIN, ORDER BY
  - Coloane selective – care întorc un result-set mic (mai puțin de 20% din înregistrările tabelului)
  - *(Selectivitatea se referă la câte linii există pentru fiecare valoarea unică a cheii de indexare)*
  - O coloană care conține doar 0 și 1 are o selectivitate slabă și nu e indicat a să fie folosită într-un index




# Indexare

- Un index poate fi *unique* sau *non-unique*
  - Un index poate folosi maxim 16 coloane pentru cheie
  - Dimensiunea combinată a coloanelor folosite pentru cheia de indexare nu poate depăși 900 octeți
  - Indecșii aduc pe lângă eficiență la interogare și costuri pentru întreținerea lor
- 



# Indexare

- Indecșii ar trebui adăugați doar dacă interogările îi pot folosi în mod eficient
  - Indecșii trebuie monitorizați – dacă nu mai sunt folosiți ar trebui eliminați
  - Prea mulți indecși pe o tabelă pot afecta performanța în momentul în care se fac actualizări – MSSQL trebuie să actualizeze și indecșii
- 


# Indexare

- Crearea unui index

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]  
    INDEX index_name  
ON {  
    [ database_name. [ schema_name ] . | schema_name. ]  
    table_or_view_name}  
( column [ ASC | DESC ] [ ,...n ] )
```




# Indexare

- Cheia primară este una singură și nu poate conține valori NULL
  - Se pot crea mai mulți indecși UNIQUE
  - Se pot include coloane ce conțin valori NULL (una singură/index)
  - Putem avea un singur index CLUSTERED și mai mulți NONCLUSTERED (259)
- 




# Indexare

- ALTER INDEX similar cu CREATE INDEX
  - Nu se pot schimba coloanele folosite și ordinea lor
  - Exemplu: index1.sql
  - La crearea unei chei primare se creează și un index
  - De regulă prima dată se creează indexul clustered
- 



# Indexare

- Pentru unicitate pe câmpuri noncheie trebuie folosită clauza UNIQUE
  - Exemplu: index2.sql
  - Se poate crea un index pe mai multe coloane atunci când acele coloane sunt folosite des ca și criterii de căutare în operații de selectare
  - Exemplu: index2.sql
  - Alegerea coloanelor pentru un index este o operație delicată
- 






# Indexare

- Recomandări pentru alegerea coloanelor pentru indecși:
  - O coloană cu selectivitate slabă (puțin valori distincte) nu e bună pentru un index
  - La un index compus coloanele cele mai selective trebui puse primele
- Database Tuning Advisor ne poate da indicații în legătură cu alegerea adecvată a indecșilor



# Indexare

- Direcția de sortare a colonei unui index poate fi stabilită folosind ASC sau DESC
  - Exemplu: index3.sql
  - Ordinea de indexare ar trebui să se potrivească cu ordinea în care utilizatorii vor folosi clauza ORDER BY în interogări – astfel performanța va fi îmbunătățită
- 

# Indexare

- Metadata pentru indecși
- Unde sunt, numele, tipul, coloanele definatorii
- Procedura stocată sistem **sp\_helpindex**
- Exemplu: EXEC sp\_helpindex  
'HumanResources.Employee'
- sys.indexes – vizualizare sistem pentru mai multe detalii
- Exemplu: index4.sql

# Indexare

- Dezactivarea unui index
- Se face pentru depanarea unui index sau în cazul unei erori de disc
- Reactivarea unui index determină recreerea indexului
- Sintaxa:


```
ALTER INDEX index_name ON  
table_or_view_name DISABLE
```

# Indexare

- Cât timp un index este dezactivat definiția sa rămâne în tabelele sistem dar nu mai poate fi folosit
- Pentru indecșii non-clustered datele sunt eliminate din baza de date
- Pentru indecșii clustered datele rămân pe disc dar nu le putem interoga
- Pentru reactivarea unui index
  - `CREATE INDEX ... DROP EXISTING`
  - `ALTER INDEX REBUILD`



# Indexare

- Eliminarea unui index
  - DROP INDEX  
    <table\_or\_view\_name>.<index\_name> [ ,...n ]
  - Indexul este eliminat fizic din baza de date
- 

# Indexare

- ALTER INDEX se folosește pentru
  - a modifica opțiuni ale unui index
  - recrea și reorganiza un index
  - dezactivarea unui index
- ALTER INDEX nu se folosește pentru
  - Adăugare
  - Ștergere
  - Rearanjarea coloanelor


# Indexare

- Pentru schimbarea coloanelor unui index se folosește `CREATE INDEX ... DROP EXISTING`
- O singură comandă elimină și recreează indexul
- Exemplu: `index5.sql`






# Indexare

- Controlarea performanței și concurenței la crearea indexului
  - Putem stabili cum să fie creat un index
  - Putem stabili folosirea unui plan de execuție paralel
  - Putem crea indexul în tempdb pentru a nu crește dimensiunea fișierului în baza de date
  - În Enterprise Edition se poate folosi clauza ONLINE ce permite acces concurent la tabelă în timpul creării indexului
- 



# Indexare


- tempdb poate fi folosită ca și bază de date în care se creează indexul
  - Baza de date sistem tempdb este folosită pentru a a gestiona
    - Conexiuni
    - Tabele temporare
    - Proceduri stocate temporare
    - Tabele temporare necesare pentru procesarea query-urilor
- 

# Indexare

- Baza de date tempdb ar putea fi izolată pe un disc separat ceea ce ar putea duce în anumite circumstanțe la creșteri ale performanței
- Dacă crearea unui index durează mult se poate folosi opțiunea SORT\_IN\_TEMPDB pentru a îmbunătăți performanța la crearea indexului (pentru tabele mari)
- Opțiunea poate fi folosită atât la CREATE INDEX cât și la ALTER INDEX
- WITH (SORT\_IN\_TEMPDB = {ON | OFF})
- Exemplu: index6.sql



# Indexare

- Controlul numărului de procesoare care rezolvă o interogare
  - În SQL Server Enterprise Edition se poate controla numărul de procesoare implicate într-o operație de indexare cu opțiunea MAXDOP
  - Paralelismul poate îmbunătăți performanța operației de indexare
  - Valoarea implicită este zero = se pot folosi oricâte procesoare
- 

# Indexare

- MAXDOP = 1 dezactivează paralelismul
- Exemplu:  
CREATE NONCLUSTERED INDEX  
NI\_Address\_AddressLine1 ON  
Person.Address (AddressLine1)  
WITH (MAXDOP = 4)
- SQL Server nu va folosi mai mult de 4 procesoare

# Indexare

- În SQL Server Enterprise putem permite accesul la tabelă în timpul operației de indexare
- Exemplu:  

```
CREATE NONCLUSTERED INDEX  
    NCI_ProductVendor_MinOrderQty ON  
    Purchasing.ProductVendor(MinOrderQty)  
    WITH (ONLINE = ON)
```
- În timpul operației de indexare nu se blochează tabela pe termen lung



# Indexare

- Opțiuni pentru indexare
  - INCLUDE – adăugare de coloane non-cheie la un index non-clustered
  - PAD\_INDEX
  - FILLFACTOR


# Indexare

- Clauza INCLUDE
- Interogare de tip *covering* (acoperitor) = toate coloanele referite se găsesc într-un index non-clustered
- Performanță sporită - MSSQL nu trebuie să scoată date din indexul clustered sau din heap
- Dezavantaj – cheia unui index poate avea max. 900 octeți și 16 coloane





# Indexare

- Cuvântul INCLUDE permite adăugare a max. 1023 coloane non-cheie la un index non-clustered
  - Astfel performanța interogărilor crește prin crearea unui index acoperitor
  - Coloanele non-cheie nu sunt stocate la fiecare nivel al indexului
  - Datele se vor regăsi doar în nodurile terminale ale indexului non-clustered
- 

# Indexare

- Sintaxa:

```
CREATE NONCLUSTERED INDEX index_name  
ON table_or_view_name ( column [ ASC | DESC ] [  
    ,...n ] )  
INCLUDE ( column [ ,... n ] )
```


- Exemplu: index7.sql

# Indexare

- PAD\_INDEX și FILLFACTOR
- Factorul de umplere (FILLFACTOR) al unui index se referă la gradul de umplere al *nivelului terminal* al paginilor indexului atunci când este creat pentru prima oară
- Implicit este zero = paginile se vor umple cât de mult este posibil
- Dacă se lasă spațiu liber se pot introduce noi înregistrări în tabelă/index fără a diviza pagini



# Indexare

- O pagină este divizată atunci când se adaugă o nouă înregistrare la o pagină de index plină
  - La divizare jumătate din înregistrări sunt mutate pe o pagină nouă
  - Divizările de pagini pot încetini operațiile INSERT
  - Pe de altă parte paginile complete oferă citire rapidă – se accesează mai puține pagini pentru a citi mai multe înregistrări
- 

# Indexare

- Sintaxa:  
WITH (PAD\_INDEX = { ON | OFF },  
FILLFACTOR = fillfactor)
- Se poate folosi la CREATE/ALTER INDEX
  - ▣ Exemplu: index8.sql
- PAD\_INDEX = ON stabilește ca să se lase spațiu liber și pe nivelurile intermediare ale indexului

# Indexare

- **FILLFACTOR = 100**
  - operații de citire foarte eficiente
  - Încetinește operațiile de scriere
- **Valori pentru FILLFACTOR**
  - 100 = pentru tabele în care nu se fac modificări
  - 80-90 = pentru tabele cu modificări puține
  - <50 = pentru tabele cu modificări frecvente ale cheilor de indexare

■

# Indexare


- Crearea unui index pe un filegroup
- În mod implicit un index este creat pe același filegroup ca și tabela

CREATE INDEX ... ON filegroup | default

- Exemplu: index9.sql



# Indexare

- Indexarea unei părți dintr-o tabelă
  - În MSSQL 2008 se pot crea indecși filtrați pentru a oferi suport interogărilor ce necesită doar o mică parte din datele din tabelă
  - Se pot indexa doar liniile care îndeplinesc un anumit criteriu
  - Se reduce spațiul pe disc ocupat de index
  - Se îmbunătățește performanța interogărilor
  - Exemplu: index10.sql
- 





# DCL

- Principals = obiecte (login, rol, aplicație) cărora li se pot da permisiuni pentru a accesa diferite obiecte din baza de date
- Securables = obiecte (de ex. tabele, vizualizări) la care se poate controla accesul
- Permisiuni = drepturi individuale ce se pot atribui/retrage la/de la un principal pentru a accesa un obiect securabil

# DCL

- Principals:

- Windows – bazați pe conturi de utilizator de domeniu, grupuri de domeniu, conturi de utilizator locale, grupuri locale. Se adaugă la SQL Server, li se dau permisiuni de acces la obiecte și au acces la MSSQL prin *Windows Authentication*
- SQL Server – login-uri la nivel de SQL Server și roluri de server fixate. Sunt independenți de orice entitate Windows. Au login name și parolă. Rolurile de server sunt grupări de permisiuni la nivel de instanță MSSQL la care alți principali pot deveni membri, moștenind permisiunile acelui rol de server




# DCL

- Principals:
  - Principali la nivel de bază de date
    - useri
    - roluri la nivel de bază de date (fixate sau definite de utilizator)
    - roluri de aplicație




# DCL

- Principali Windows
  - În MSSQL putem crea login de tip Windows pe baza conturilor de utilizator sau grup Windows
  - Un login Windows poate fi asociat cu
    - Un user de domeniu
    - Un user local
    - Un grup Windows
- 




# DCL

- Login-urile de tip Windows nu mai au nevoie de o altă parolă
  - Autentificarea este rezolvată de SO
  - Contul curent de utilizator trebuie să fie identificat ca un login de instanța SQL Server sau contul curent trebuie să aparțină unui grup Windows care există ca și login pe MSSQL
- 



# DCL

- Login-urile Windows acționează doar la nivelul SO.
  - Nu le putem atribui acces la anumite obiecte
  - Va trebui să creăm utilizatori la nivelul bazei de date pe care să-i asociem cu un login – drepturile de acces la diverse obiecte se vor da user-ilor bazei de date
  - Autentificarea Windows este mai strictă – se rezolvă la nivelul SO – parolele nu se transmit prin rețea
- 

# DCL

- Sintaxa:

```
CREATE LOGIN login_name
FROM WINDOWS
[ WITH DEFAULT_DATABASE = database
| DEFAULT_LANGUAGE = language
]
| CERTIFICATE certname
| ASYMMETRIC KEY asym_key_name
```

- Exemplu: DCL1.sql

# DCL

- Pentru a vizualiza login-urile existente putem interoga vizualizarea *sys.server\_principals*

```
SELECT name, sid
```

```
FROM sys.server_principals
```

```
WHERE type_desc IN ('WINDOWS_LOGIN',  
                    'WINDOWS_GROUP')
```

```
ORDER BY type_desc
```

- Exemplu: DCL2.sql



# DCL


- ALTER LOGIN
  - Se poate schimba baza de date implicită
  - Se poate schimba limba implicită
  - Activa/dezactiva un login
- Sintaxa: (*Books Online*)  
ALTER LOGIN login\_name  
{  
ENABLE | DISABLE  
|  
WITH  
| DEFAULT\_DATABASE = database  
| DEFAULT\_LANGUAGE = language }  
Exemplu: DCL4.sql

# DCL

- DROP LOGIN
- Se elimină login-ul din MSSQL
- Dacă este conectat orice acțiune ulterioară nu este permisă
- Sintaxa: DROP LOGIN login\_name
- Dacă login-ul deține obiecte securabile comanda DROP va eșua



# DCL


- Interzicerea accesului la MSSQL
  - DENY CONNECT
  - Exemplu: DCL3.sql
- 

# DCL

- Principali SQL
- La autentificare Windows
  - SO efectuează autentificarea
  - MSSQL efectuează autorizarea (ce acțiuni poate întreprinde un user autentificat)
- La autentificarea SQL
  - MSSQL face atât autentificarea cât și autorizarea
- Login-urile SQL sunt create în server și parolele sunt stocate la nivel de server




# DCL

- Autentificarea SQL este mai puțin sigură pentru că sunt implicate în mod implicit parole
  - Se folosește pentru aplicații de la terți și aplicații care nu rulează în mod obligatoriu pe sisteme Windows.
  - Sunt implementate politici de parole: dimensiune, complexitate, timp de expirare etc.
- 



# DCL

- Login-urile SQL operează doar la nivelul serverului de baze de date
  - Nu le putem da drepturi de acces la diverse obiecte ale bazelor de date de pe server
  - Trebuie creat user la nivelul bazei de date asociat login-ului pentru a putea lucra cu obiecte ale bazelor de date
- 

# DCL

- SQL Server suportă principali bazați pe:
  - Login individual
  - Roluri de server – mai mulți useri se pot asocia la rol

- Sintaxa:

```
CREATE LOGIN login_name  
[WITH PASSWORD = ' password ' [ HASHED ] [  
    MUST_CHANGE ],  
SID = sid], DEFAULT_DATABASE = database,  
DEFAULT_LANGUAGE = language,  
CHECK_EXPIRATION = { ON | OFF },  
CHECK_POLICY = { ON | OFF },  
CREDENTIAL = credential_name ]
```

Exemplu: DCL5.sql

# DCL

- Metainformații despre login-uri SQL
- Din vizualizarea sistem `sys.server_principals`
- Exemplu: `DCL6.sql`
- Modificarea unui login SQL Server se face cu comanda `ALTER LOGIN`
- Se poate modifica:
  - Parola, baza de date implicită, limba, numele login-ului, politica de parole, activare/dezactivare



# DCL

- Sintaxa:

```
ALTER LOGIN login_name
```

```
{ ENABLE | DISABLE
```

```
|
```

```
WITH PASSWORD = 'password' [ OLD_PASSWORD =  
    'oldpassword '
```

```
| [ MUST_CHANGE | UNLOCK ] ] | DEFAULT_DATABASE =  
    database
```

```
| DEFAULT_LANGUAGE = language
```

```
| NAME = login_name | CHECK_POLICY = { ON | OFF }
```

```
| CHECK_EXPIRATION = { ON | OFF }
```


```
| CREDENTIAL = credential_name | NO CREDENTIAL
```

```
}
```

- Exemplu: DCL7.sql




# DCL

- Funcția LOGINPROPERTY întoarce informații despre login și setările politicilor de parole și starea lor
  - Exemplu: DCL8.sql
  - Eliminarea unui login se face cu DROP LOGIN
  - Un login nu poate fi eliminat dacă deține obiecte securabile pe server
- 



# DCL


- Gestiunea membrilor rolurilor de server
  - Rolurile de server fixate sunt grupuri SQL predefinite care au atribuite permisiuni la nivel de server (spre deosebire de permisiunile la nivel de bază de date sau schemă)
  - Nu putem crea astfel de roluri
  - Putem adăuga doar login-uri (Windows sau SQL) la aceste roluri
- 

# DCL

- Rolul sysadmin este rolul cu cel mai mare nivel de permisiuni
- Adăugarea/eliminarea unui login la un rol fixat de server se face cu procedura stocată sistem *sp\_addsrvrolemember*, *sp\_dropsrvrolemember*
- Sintaxa:  
`sp_addsrvrolemember [ @loginame= ] 'login',  
[ @rolename = ] 'role'`
- Exemplu: DCLg.sql



# DCL

- Un login ar trebui să primească permisiunile minime de care are nevoie pentru a-și rezolva problema
  - Adăugarea unui login la rolul sysadmin este o acțiune care trebuie analizată/cântărită cu multă atenție
  - Rolurile de server fixate definesc un grup de permisiuni la nivel de SQL Server
- 




# DCL

- La fel ca și login-urile rolurile de server au un identificador de securitate ce poate fi interogată din `sys.server_principals`
- Rolurile de server pot avea membri care moștenesc permisiunile rolului
- Exemplu: DCL10.sql
- Permisiunile rolurilor de server: Books online



# DCL

- Principali la nivel de bază de date = obiecte ce reprezintă useri cărora li se pot atribui permisiuni de acces la baza de date sau obiecte ale bazei de date
  - Login-urile operează la nivelul serverului (conectare)
  - Principalii de baze de date operează la nivelul bazei de date (selectare date, manipulare, operații DDL, gestionare permisiuni la nivel de bază de date)
- 


# DCL

- Principali la nivel de bază de date
  - Useri = contextul de securitate la nivel de bază de date sub care se execută cererile în cadrul bazei de date – sunt asociați cu login-uri Windows sau SQL
  - Roluri de bază de date = fixate la nivel de bază de date și definite de utilizator (permit gestionarea mai facilă a permisiunilor față de atribuirea drepturilor în mod individual la fiecare user)
  - Roluri de aplicație = grupări de permisiuni ce nu permit membri. Ne putem conecta cu rolul de aplicație. Se anulează toate permisiunile pe care le are login-ul și vor fi active doar permisiunile atribuite rolului





# DCL

- Creare de useri în baza de date
  - O dată creat un login poate fi asociat cu un user din baza de date
  - Un login poate fi asociat cu mai multe baze de date
  - Un login poate fi asociat cu un singur user dintr-o bază de date
- 

# DCL

- Sintaxa:

```
CREATE USER user_name
```

```
[ FOR
```

```
    { LOGIN login_name
```

```
    | CERTIFICATE cert_name
```

```
    | ASYMMETRIC KEY asym_key_name
```

```
    }
```

```
]
```

```
[ WITH DEFAULT_SCHEMA = schema_name ]
```


```
Exemplu: DCL11.sql
```

# DCL

- Informații despre useri pe conexiunea curentă se pot obține cu procedura stocată sistem *sp\_helpuser*
- Sintaxa:  
`sp_helpuser [ [ @name_in_db= ] 'security_account' ]`
- `EXEC sp_helpuser 'Veronica'`
- Modificarea unui user (nume sau schemă)  
`ALTER USER user_name  
WITH NAME = new_user_name  
| DEFAULT_SCHEMA = schema_name`




# DCL

- Eliminarea unui user  
DROP USER user\_name
  - Nu putem elimina un user care este proprietar al unor obiecte
- 



# DCL

- Rezolvarea unor useri orfani din baza de date
  - La migrarea unei baze de date pe un nou server (prin BACKUP/RESTORE) relația dintre login și user se poate întrerupe
  - Un login are un identificador de securitate (sid) care îl identifică în mod unic pe instanța MSSQL
  - sid-ul este stocat pentru userul asociat loginului în fiecare bază de date în care login-ul are acces.
- 

# DCL

- La crearea unui nou login cu același nume pe un alt server nu i se va atribui același sid (excepție este cazul în care se precizează sid-ul cu comanda CREATE LOGIN)
- Pentru căutarea unor useri orfani putem folosi următorul exemplu:
  - Exemplu: DCL12.sql
- La restaurarea unei baze de date pe un server nou user-ul nu va avea login asociat prin urmare va deveni user orfan.



# DCL

- Comanda ALTER USER are opțiunea WITH LOGIN care poate reface asocierea dintre un login și un user
- Este valabil atât pentru login-uri SQL cât și pentru login-uri Windows
- Dacă un user din Active Directory a fost șters/recreat comanda este utilă



# DCL

- Roluri fixate la nivel de bază de date
- Fiecare bază de date are roluri fixate care au atribuite permisiuni (selectare, creare etc.)
- Rolurile fixate ale bazei de date au membri
- Membri rolului moștenesc drepturile rolului
- EXEC sp\_helpdbfixedrole
- EXEC sp\_helprolemember



# DCL

- Gestionarea rolurilor fixate la nivel de bază de date
- Asocierea unui user cu un rol al bazei de date se face cu procedura stocată *sp\_addrolemember*
- Sintaxa:  


```
sp_addrolemember [ @rolename = ] 'role',  
[ @membername = ] 'security_account'
```

# DCL

- Eliminarea unui user dintr-un rol  
sp\_droprolemember [ @rolename= ] 'role',  
[ @membername= ] 'security\_account'
  - Exemplu: DCL14.sql



# DCL


- Roluri definite de utilizator în baza de date
  - Permisele la obiecte securabile pot fi gestionate mai ușor cu roluri definite de utilizator
  - Drepturile nu trebuie atribuite fiecărui user în parte
  - Permisele la obiecte securabile vor fi atribuite rolului
  - Membri rolului vor moșteni toate permisiunile
- 

# DCL

- Un nou rol se creează cu comanda CREATE ROLE
- Sintaxa:
  - ▣ CREATE ROLE role\_name [ AUTHORIZATION owner\_name ]
- owner\_name este proprietarul (user sau alt rol) rolului care poate gestiona rolul
- EXEC sp\_helprole
- ALTER ROLE role\_name WITH NAME = new\_name (schimbă numele unui rol)
- Exemplu: DCL15.sql



# DCL

- Roluri de aplicație = hibrid între login și rol de bază de date
  - Poate primi drepturi la fel ca un rol la nivel de bază de date
  - Rolul de aplicație nu permite membri
  - Este activat printr-o procedură stocată sistem cu parolă
  - Rolul aplicație elimină toate drepturile pe care le are login-ul
- 

# DCL

- Rolul de aplicație nu are membri prin urmare are nevoie de o parolă pentru ca permisiunile să fie activate

- Sintaxa pentru crearea unui rol aplicație

```
CREATE APPLICATION ROLE
```

```
    application_role_name
```

```
    WITH PASSWORD = ' password ' [,
```

```
        DEFAULT_SCHEMA = schema_name ]
```

- Exemplu: DCL16.sql

# DCL

- Procedura stocată sistem *sp\_setapprole* este folosită pentru a activa permisiunile rolului aplicație pentru sesiunea curentă
- Chiar dacă login-ul s-a făcut cu permisiuni de sysadmin, procedura va determina ca să fie active doar permisiunile rolului
- Pentru a reveni la permisiunile anterioare trebuie închisă conexiunea și deschisă o nouă conexiune

# DCL

- Putem modifica numele, parola sau schema implicită a unui rol de aplicație

```
ALTER APPLICATION ROLE application_role_name
```

```
WITH NAME = new_application_role_name
```


```
| PASSWORD = ' password '
```

```
| DEFAULT_SCHEMA = schema_name
```





# DCL


- Rolurile de aplicație sunt utile pentru dezvoltatorii de aplicații care doresc să aibă drepturi de acces doar prin aplicație
  - Conexiunea la server se poate realiza și cu alte aplicații (Access, MSSQL Management Studio)
  - Pentru a preveni acest lucru permisiunile login-ului se restricționează și folosim un rol la nivel de aplicație pentru a stabili drepturile necesare. Astfel user-ul poate accesa baza de date doar prin aplicație – aplicație care este programată să folosească rolul de aplicație
- 

# Obiecte securabile

- Obiectele securabile sunt resurse la care MSSQL controlează accesul prin intermediul permisiunilor
- Avem trei niveluri ierarhice pentru obiectele securabile:
  - Nivelul server (login-uri, baze de date, endpoint)
  - Nivelul bazei de date (useri, roluri, certificate, scheme)
  - Nivelul schemei (schema, tabele, vizualizări, proceduri, funcții)



# Obiecte securabile


- Permisele permit unei entități să efectueze acțiuni asupra obiectelor securabile
  - Comenzile pentru controlul accesului unui principal la obiecte securabile sunt GRANT, REVOKE, DENY – indiferent de nivelul la care se află obiectul securabil
- 

# Obiecte securabile

- În MSSQL 2008 a fost introdus SQL Server Audit Object – care permite colectarea de acțiuni la nivel de server sau bază de date pe care dorim să le monitorizăm
- Informația de audit poate să ajungă în:
  - Fișier
  - Windows Application event log
  - Windows Security event log



# Obiecte securabile

- Permisele se aplică obiectelor la cele trei niveluri: server, bază de date și schemă
  - Există o listă de nume de permisiuni care sunt aplicate la diverse obiecte securabile și care implică diferite niveluri de autorizare față de obiectul securabil
- 

# Obiecte securabile

- Permisele importante:
- ALTER = permite celui ce are permisiunea să utilizeze comenzile ALTER, CREATE, DROP. De ex. folosirea comenzii ALTER TABLE necesită permisiunea ALTER pe tabela respectivă
- AUTHENTICATE = permite celui ce are această permisiune să fie de încredere la nivel de server sau bază de date

# Obiecte securabile

- CONNECT = permite conectarea la o instanță de MSSQL sau la un endpoint
- CONTROL = toate permisiunile sunt activate atât pentru obiect cât și pentru toate obiectele încuibate (de ex. permisiunea CONTROL pe o schemă va implica aceeași permisiune pe toate tabelele, vizualizările și alte obiecte din schemă)


# Obiecte securabile

- CREATE = permite celui ce are această permisiune să creeze obiecte securabile
- IMPERSONATE = permite impersonarea altui principal (login sau user). De ex. folosirea lui EXECUTE AS pentru un login necesită această permisiune.
- TAKE OWNERSHIP = permite celui ce are această permisiune să devină proprietarul obiectului securabil pe care are această permisiune.





# Obiecte securabile

- VIEW = permite vizualizarea metadatelor în legătură cu obiectul securabil
  - Pentru a vedea permisiunile disponibile pe MSSQL există funcția sistem `sys.fn_built_in_permissions`
- 

# Obiecte securabile

- Sintaxa:

sys.fn\_builtin\_permissions


( [ DEFAULT | NULL ] | empty\_string |  
APPLICATION ROLE | ASSEMBLY | ASYMMETRIC KEY |  
CERTIFICATE | CONTRACT | DATABASE |  
ENDPOINT | FULLTEXT CATALOG | LOGIN |  
MESSAGE TYPE | OBJECT | REMOTE SERVICE  
BINDING |  
ROLE | ROUTE | SCHEMA | SERVER | SERVICE |  
SYMMETRIC KEY | TYPE | USER | XML SCHEMA  
COLLECTION )

# Obiecte securabile

- Permisele ce pot fi atribuite în MSSQL
- Exemplu: DCL17.sql
- covering permission = numele clasei de permisiuni care este mai sus în ierarhia de permisiuni încuibate




# Obiecte securabile

- Obiecte securabile și permisiuni la nivel de server
  - Obiectele securabile la nivel de server sunt unice într-o instanță MSSQL – login-uri, endpoint, baze de date
  - Permisele pentru securabile la nivel de server pot fi atribuite doar principalilor la nivel de server (login SQL Server sau Windows)
- 



# Obiecte securabile

- Entitatea care primește permisiuni la nivel de server poate crea baze de date, crea login-uri, poate crea linked-servers
  - Permisele la nivel de server permit oprirea instanței MSSQL sau folosirea SQL Profiler
  - Grantor – cel care dă permisiunea
  - Grantee – cel ce primește permisiunea
- 

# Obiecte securabile

- Sintaxa:  
GRANT Permission [ ,...n ]  
TO grantee\_principal [ ,...n ]  
[ WITH GRANT OPTION ]  
[ AS grantor\_principal ]
- Pentru a interzice permisiuni unui principal asupra unui obiect securabil folosim comanda DENY

# Obiecte securabile

- Sintaxa:  
DENY permission [ ,...n ]  
TO grantee\_principal [ ,...n ]  
[ CASCADE ]  
[ AS grantor\_principal ]
- permission = una sau mai multe permisiuni la nivel de server
- CASCADE – permisiunea va fi eliminată și de la toți principalii la care grantee-ul a dat mai departe permisiunea

# Obiecte securabile

- Revocarea unei permisiuni se face cu comanda REVOKE
- Revocarea elimină o permisiune care a fost atribuită cu GRANT sau interzisă cu DENY
- Sintaxa:  
REVOKE [ GRANT OPTION FOR ] permission [ ,...n ]  
FROM < grantee\_principal > [ ,...n ]  
[ CASCADE ]  
[ AS grantor\_principal ]
- Exemplu: DCL18.sql, DCL19.sql





# Obiecte securabile



# Obiecte securabile



# Obiecte securabile



# Obiecte securabile



# Obiecte securabile



# Obiecte securabile