# POGLAVLJE 5

# Proceduralno programiranje

**U OVOM POGLAVLJU RAZMATRA SE TRANSACT-SQL** kao proceduralni programski jezik i opisuju *proceduralni objekti*. Razumevanje proceduralne strukture, koja vam stoji na raspolaganju u Transact-SQL jeziku, predstavlja osnovu za kreiranje snimljenih procedura i korisnički definisanih funkcija.

SQL Server obezbeđuje dosta funkcionalnosti vezanih za proceduralno programiranje. Prvo ćemo razmatrati naredbe kojima se definišu različite proceduralne strukture. Naučićete nešto više o uslovnoj logici, kotroli toka, rukovanju izuzetaka i još mnogo toga. Naučićete sintaksu različitih proceduralnih struktura, kao i razloge zbog kojih se koriste SQL Server objekti. Spremite se da kreirate programe korišćenjem Transact-SQL jezika.

U ovom poglavlju naučićete kako da:

- Koristite proceduralne strukture u Transact-SQL skriptovima
- Kreirate snimljene procedure
- Definišete poglede
- Samostalno definišete funkcije
- Kreirate sinonime

## Proceduralne strukture

U ovom odeljku razmatraćemo nizove naredbi, elemente jezika za upravljanje tokom, upravljanje greškama i osnovnu obradu transakcija. Svi ovi elementi se koriste prilikom kreiranja snimljenih procedura, korisnički definisanih funkcija i trigera.

### Nizovi naredbi

*Nizovi naredbi* predstavljaju skupove Transact-SQL naredbi, koje se prosleđuju zajedno SQL Serveru u cilju izvršavanja. Svaki niz naredbi koji se prosledi do SQL Servera prevodi se u odgovarajući plan izvršavanja, a zatim se taj plan i praktično realizuje.

Sa stanovišta aplikacije, svaki skup naredbi koje se izvršavaju smatra se izdvojenim nizom naredbi. Uopšteno rečeno, svaki put kada aplikacija izvršava SQL naredbe, taj niz naredbi tretira se kao posebna celina.

Transact-SQL skript može da sadrži jedan ili više nizova naredbi. Prilikom kreiranja skriptova, možete da uključite više nizova naredbi izdvajanjem Transact-SQL blokova pomoću GO ključne reči. Ključna reč GO definiše kraj niza naredbi za onaj alat koji izvršava skript; samu GO naredbu ne izvršava SQL Server. Svaki put kada vidite liniju koja počinje sa dve crtice, znajte da se radi o linijskom komentaru. Razmotrimo sledeći primer:

(Strana 134 – kod primera)

```
USE SandboxDB;
  Prvi niz naredbi
CREATE TABLE T1 (C1 int NOT NULL);
INSERT INTO T1 VALUES (1);
GO
  Drugi niz naredbi
CREATE TABLE T2 (C2 int NOT NULL);
INSERT INTO T2 VALUES (2);
```

**SANDBOXDB**

Svi primeri navedeni u ovom poglavlju koriste praznu bazu podataka SandBoxDB. Ova baza podataka je prethodno korišćena u Poglavlju 3. Možete da koristite postojeću bazu podataka ukoliko ste je prethodno kreirali, ili možete da kreirate novu bazu podataka izvršavanjem sledeće naredbe:

```
CREATE DATABASE SandboxDB;
```

Prilikom izvršavanja prethodnog primera, prvi niz naredbi se prosleđuje do servera, prevodi i izvršava. Nakon što se završi izvršavanje prvog niza naredbi, drugi niz naredbi se prosleđuje do servera, prevodi i izvršava.

Postoji nekoliko pravila koja moraju da se poštuju prilikom izvršavanja niza naredbi.

- Neke naredbe moraju da se izvršavaju nezavisno od drugih naredbi, a to su `CREATE PROCEDURE`, `CREATE VIEW`, `CREATE FUNCTION`, `CREATE DEFAULT`, `CREATE RULE`, `CREATE SCHEMA` i `CREATE TRIGGER`.
- Sve promenljive moraju biti definisane i korišćene u jednom istom nizu naredbi.
- Komentari koji se prostiru na nekoliko linija, a koji se označavaju sa /* i */ moraju da počnu i završe se u istom nizu naredbi.
- Struktura tabele ne sme da se menja, odnosno nove kolone ne smeju da se referenciraju u istom nizu naredbi.

Na osnovu grešaka prevođenja koje se dobijaju prilikom izvršavanja niza naredbi, kao što su sintaksne greške, prekida se dalje izvršavanje niza naredbi. Uopšteno rečeno, greška prilikom izvršavanja zaustavlja izvršavanje tekuće naredbe, a samim tim onemogućava izvršavanje ostalih naredbi koje se nalaze u nizu naredbi iza naredbe čije je izvršavanje prekinuto. To znači da se niz naredbi ne može izvršavati parcijalno prilikom pojave greške u toku njegovog izvršavanja. Neke greške koje se javljaju prilikom izvršavanja će zaustaviti izvršavanje samo trenutne naredbe, a omogućiti izvršavanje ostalih naredbi koje se nalaze u nizu naredbi. Narušavanje ograničenja je primer takvih grešaka.

Počevši od SQL Server 2005 verzije, koristi se prevođenje na nivou naredbi. To uzrokuje nešto drugačije funkcionisanje u odnosu na prethodne SQL Server verzije. Razmotrimo sledeći primer:

```
USE SandboxDB;
DROP TABLE T1;
GO
CREATE TABLE T1 (C1 int NOT NULL);
INSERT INTO T1 VALUES (1);
INSERT INTO T1 VALUES (2,2);
INSERT INTO T1 VALUES (3);
GO
SELECT * FROM T1;
```

U SQL Server 2005 verziji i novijim verzijama, funkcionisanje je sledeće. Pošto tabela T1 trenutno ne postoji, prevodi se `CREATE TABLE` naredba, ali ne i `INSERT` naredbe. Nakon izvršavanja `CREATE TABLE` naredbe, prevode se i izvršavaju `INSERT` naredbe, jedna po jedna. Pošto druga `INSERT` naredba definiše veliki broj vrednosti, ona uzrokuje pojavu greške prilikom izvršavanja, i time prekida dalje izvršavanje ovog niza naredbi. `SELECT` naredba vraća samo jednu vrstu.

Slično, u SQL Server 2000 verziji i verzijama koje su joj prethodile, proces se odvija na identičan način; međutim, sve `INSERT` naredbe se prevode i izvršavaju odjednom. Zbog greške u drugoj naredbi, niz naredbi se neće u potpunosti izvršiti, ali nijedan podatak neće biti unet u bazu podataka. To znači da se izvršavanjem `SELECT` naredbe neće prikazati nijedna kolona.

## Promenljive

Promenljiva predstavlja kontejner za jednu vrednost podatka određenog tipa. U Transact-SQL jeziku, sve promenljive se označavaju početnim `@` simbolom. Moguće je definisati maksimalno 10 hiljada promenljivih u jednom nizu naredbi koje se istovremeno izvršavaju.

Lokalne promenljive se u Transact-SQL skriptovima koriste za:

- Skladištenje vrednosti koje će se testirati u upavljačkim naredbama
- Brojače u petljama
- Smeštanje rezultata izvršavanja nekog izraza
- Očitavanje polja vrednosti za jedan zapis korišćenjem `SELECT` naredbe

Promenljive se koriste i za prosleđivanje vrednosti parametara snimljenih procedura i korisnički definisanih funkcija.

Prilikom deklarisanja promenljive, neophodno je da navedete njen naziv, tip podataka, a u nekim situacijama i dužinu i preciznost tipa podataka. Opciono, možete da koristite reč AS prilikom deklarisanja promenljivih. `DECLARE` naredba se može koristiti prilikom deklarisanja većeg broja promenljivih, pri čemu se pojedinačne promenljive odvajaju zarezima.

```
DECLARE @Var1 int;
DECLARE @Var2 as int;
DECLARE @Var3 varchar(25),
  @Var3 decimal(5,2)
```

U prethodnom primeru deklarisane su tri promenljive korišćenjem dve `DECLARE` naredbe. Tip podataka varchar zahteva da navedete dužinu promenljive. Tip podataka decimal zahteva da navedete dužinu i preciznost. Promenljiva `@Var3` će se koristiti za smeštanje decimalne vrednosti, uz korišćenje ukupno pet cifara, od kojih se dve nalaze desno od decimalnog zareza (odnosno decimalne tačke).

Preporučeni način za definisanje sadržaja promenljive je korišćenje SET naredbe. Postoji mogućnost da koristite i `SELECT` naredbu za dodeljivanje vrednosti jednoj promelljivoj ili većem broju promenljivih istovremeno. Prilikom korišćenja SET naredbe, možete da definišete vrednost samo jedne promenljive. Da biste definisali vrednost većeg broja promenljivih, neophodno je da navedete nekoliko SET naredbi, kao što je prikazano u sledećem primeru.

```
SET @Var1 = 5;
SET @Var2 = 'A varchar string';
SELECT @Var2 = 'Another varchar string',
  @Var3 = 123.45;
```

Možete da definišete vrednost promenljive, tako da bude polje konkretnog zapisa u SELECT naredbi. Vodite računa kada to radite, pošto se može dogoditi da više zapisa bude vraćeno kao rezultat izvršavanja upita, i tada će samo vrednosti poslednjeg zapisa biti smeštene u odgovarajuće promenljive.

```
USE SandboxDB;
DECLARE @CustName varchar(50);
SELECT @CustName = CustomerName
 FROM Customer
 WHERE CustomerID = 1;
```

Lokalne promenljive imaju domet od tačke u kojoj su definisane, pa sve do kraja trenutnog niza naredbi. Ukoliko pokušate da koristite promenljivu izvan predviđenog dometa javiće se greška, kao što je prikazano u sledećem primeru:

```
DECLARE @Test int;
SET @Test = 5;
GO
-- Prilikom izvršavanja sledeće naredbe pojavljuje se greška.
PRINT @Test;
```

#### POSTOJE LI GLOBALNE PROMENLJIVE?

Iako se naredbe oblika @@Version ponekad nazivaju globalnim promenljivama, to su u suštini sistemske funkcije. Ne postoji ništa slično globalnim promenljivama u Transact-SQL jeziku.

## Upravljačke naredbe

U Transact-SQL jeziku, upravljačke strukture utiču na redosled izvršavanja naredbi na osnovu vrednosti bulovih (logičkih) izraza. Ove naredbe se veoma često koriste u snimljenim procedurama i funkcijama.

## BEGIN...END

Grupe naredbi koje se koriste sa IF, WHILE i CASE naredbama mora da budu grupisane korišćenjem BEGIN i END naredbi. Svaka BEGIN naredba mora da ima odgovarajuću END naredbu u istom nizu naredbi.

## IF...ELSE

IF naredbe izračunavaju vrednost bulovog izraza i usmeravaju izvršenje na osnovu dobijenog rezultata.

```
IF @@ERROR <> 0
BEGIN
PRINT 'Javila se greška u prethodnoj naredbi.';
RETURN;
END
ELSE
PRINT 'Nema grešaka u prethodnoj naredbi.';
```

U prethodnom primeru proverava se da li promenljiva @@ERROR ima vrednost koja je različita od nule. Ukoliko je uslov ispunjen, izvršava se naredba ili blok naredbi koje se nalaze neposredno iza if. Ukoliko uslov nije ispunjen, i ukoliko postoji ELSE blok, izvršava se taj blok. U bilo kojoj situaciji u kojoj

je neophodno da se izvrši više od jedne naredbe kao odgovor na IF ili ELSE, neophodno je da se primenjuje BEGIN i END konstrukcija.

## WHILE

WHILE omogućava izvršavanje naredbe sve dok odgovarajući izraz ima vrednost True. (Ukoliko je neophodno da se izvršava više od jedne naredbe, neophodno je da primenite BEGIN i END konstrukciju). U sledećem primeru prikazano je korišćenje WHILE naredbe za potrebe realizovanja brojača:

```
DECLARE @Counter int;
SET @Counter = 1;
WHILE (@Counter <= 10)
BEGIN
  PRINT @Counter;
  SET @Counter = @Counter + 1;
END
```

WHILE naredba se može koristiti zajedno sa EXISTS ključnom reči ukoliko postoje bilo koji rezultati koji se dobijaju prilikom izvršavanja upita. To je prikazano u sledećem primeru:

```
WHILE EXISTS (SELECT * FROM T1 WHERE C1 = 1)
BEGIN
— Izvršiti neke operacije nad vrstama T1 tabele, za koje važi da je C1 = 1
END
```

## CASE

CASE naredba se koristi za zamenu vrednosti kolone u SELECT naredbi na osnovu jednog ili više izraza. U Transact-SQL jeziku, CASE naredba se korsiti za obradu elemenata po vrstama u SELECT naredbi; ona se ne koristi za evaluaciju na nivou naredbi, kao što je situacija u nekim drugim programskim jezicima.

CASE naredba se može koristiti svaki put kada je kolona omogućena u SELECT naredbi. Ona se obično koristi prilikokm zamene identifikatora ili kodova deskriptivnijim vrednostima, ali može biti korisna u mnogim drugim situacijama.

Sledeći primer vezan za AdventureWorks2008 bazu podataka demonstrira korišćenje veoma jednostavne CASE naredbe, kojom se zamenjuju nazivi odeljenja identifikatorima od dva karaktera za svako pojedinačno odeljenje.

```
USE AdventureWorks2008;
GO
SELECT Name,
 CASE Name
  WHEN 'Human Resources' THEN 'HR'
  WHEN 'Finance' THEN 'FI'
  WHEN 'Information Services' THEN 'IS'
  WHEN 'Executive' THEN 'EX'
  WHEN 'Facilities and Maintenance' THEN 'FM'
 END AS Abbreviation
FROM AdventureWorks2008.HumanResources.Department
WHERE GroupName = 'Executive General and Administration';
```

CASE naredba se može koristiti i u bulovim izrazima za pretraživaje. Ovi izrazi su veoma značajni, jer ukoliko neki od zapisa ispunjava uslov, taj zapis će se koristiti, i neće se analizirati nijedan dodatni izraz. Sledi primer korišćenja CASE naredbe u izrazu za pretraživanje:

```
USE AdventureWorks2008;
GO
SELECT ProductNumber,
       Name,
  CASE
   WHEN ListPrice = 0 THEN 'Not For Resale'
   WHEN ListPrice < 49 THEN 'Under $50'
   WHEN ListPrice BETWEEN 50 and 499 THEN '$50 - $499'
   WHEN ListPrice BETWEEN 500 and 1000 THEN '$500 - $1000'
   ELSE 'Over $1000'
  END As PriceRange
FROM AdventureWorks2008.Production.Product
ORDER BY ProductNumber;
```

## Upravljanje greškama

Očigledno je da će kad tad doći do grešaka prilikom izvršavanja Transact-SQL naredbi. Veoma je važno da razumete različite tipove grešaka i opcije koje su vam raspoložive za upravljanje greškama.

**Sintaksne greške** Sintaksne greške se javljaju veoma često i njihov uzrok su greške prilikom unosa podataka putem tastature. Kada se detektuje sintaksna greška, ne izvršava se nijedna naredba u nizu naredbi, pošto se greška detektuje u toku kompilacije, pre nego što počne izvršavanje naredbi. Veoma je važno da zapamtite da ne možete da odgovarate na pojavu grešaka putem koda; jedino što možete da uradite je da ispravite sintaksnu grešku, a zatim ponovo probate da izvršite niz naredbi.

```
USE AdventureWorks2008;
PRINT 'Does not run.';
SELECT ** FROM HumanResources.Employee;
PRINT 'Also does not run.';
```

U prethodnom primeru, PRINT naredba se nikada ne izvršava, pošto se detektuje sintaksna greška prilikom kompiliranja upita. SQL Server generiše sledeću poruku o pojavi greške:

```
Msg 102, Level 15, State 1, Line 2
Incorrect syntax near '*'.
```

**Greške u toku izvršavanja** U toku izvršavanja upita mogu se javiti greške iz više razloga. Ove greške se javljaju prilikom izvršavanja koda, nakon što se generiše plan izvršavanja. Greške prilikom izvršavanja se mogu tretirati pomoću logike za rukovanje greškama. Ova logika može da se nalazi u aplikaciji koja izvršava naredbu, ali može biti napisana i korišćenjem Transact-SQL naredbi. Način na koji se rukuje greškama u aplikaciji zavisi od API interfejsa baze podataka, koji se koristi prilikom izvršavanja upita. Ukoliko koristite ADO. NET, imate na raspolaganju SqlException klasu, putem koje možete da očitate sve detalje o greškama koje se javljaju. Više informacija o ADO.NET tehnologiji možete da pronađete u poglavlju 17, "SQL Server i .NET klijent".

```
PRINT 'Before Error';
SELECT 1/0;
PRINT 'After Error';
```

Izvršavanjem prethodnog koda dolazi do deljenja nulom, i to u drugoj naredbi. Međutim, izvršavaju se sve tri naredbe, kao što je prikazano u rezultatu izvršavanja:

```
Before Error
Msg 8134, Level 16, State 1, Line 2
Divide by zero error encountered.
After Error
```

#### Poruke o pojavama grešaka

Poruke o pojavama grešaka, koje generiše SQL Server, smeštene su u sys.messages katalogu. Kada se pojavi određena greška, ona se opisuje na osnovu sledećih svojstava:

**Broj greške** To je broj greške, onako kako je definisan u sys.messages katalogu. Greške čiji su brojevi preko 50000 su korisnički definisane greške. Brojevi do 50000 su rezervisani za sistemske greške.

**Nivo ozbiljnosti** Broj koji reprezentuje ozbiljnost greške koja se pojavila.

**Stanje** Broj koji predstavlja gde se pojavila greška. Ukoliko se jedna ista greška detektuje na više lokacija u nekom delu koda, neophodno je da se koriste različita stanja.

**Broj linije** Broj linije u kome bi trebalo da se pojavila greška.

**Poruka o pojavi greške** Poruka koja opisuje grešku koja se pojavila. Odgovarajući parametri se koriste ovde.

Karakteristike grešaka koje su dostupne putem koda zavise od metoda koji primenjujete za detektovanje greške. Ukoliko primenjujete `TRY…CATCH` konstrukciju, sve informacije o greškama se mogu očitavati u `CATCH` bloku. Ukoliko koristite `@@ERROR` sistemsku funkciju, možete da detektujete samo broj greške, i taj broj se vezuje za prvu sledeću naredbu koja treba da se izvrši nakon one koja je izazvala grešku.

#### Rukovanje greška korišćenjem TRY...CATCH konstrukcije

`TRY...CATCH` konstrukcija je uvedena u SQL Server 2005 verziji za potrebe rukovanja greškama korišćenjem Transact-SQL koda. Naredbe koje treba da se testiraju u cilju utvrđivanja grešaka nalaze se u `BEGIN TRY…END TRY` bloku. `CATCH` blok se nalazi neposredno iza `TRY` bloka i u njemu se nalazi logika za rukovanje greškama. U sledećem primeru prikazana je osnovna sintaksa:

```
BEGIN TRY
— Kod koji može da generiše greške
END TRY
BEGIN CATCH
— Logika za rukovanje greškama
END CATCH;
```

SQL Server sekvencijalno analizira svaku naredbu koja se nalazi u `TRY` bloku. Ukoliko se javi greška u toku izvršavanja, kontrola se automatski prenosi do `CATCH` bloka, potom se očitavaju informacije o greškama, a zatim loguju i prikazuju korisniku.

Unutar `CATCH` bloka, raspoloživo je nekoliko funkcija za očitavanje informacija o greškama koje se javljaju (one su prikazane u tabeli 5.1). Veoma je važno da uočite da će ove funkcije vratiti `NULL` vrednost izvan dometa `CATCH` bloka.

**Tabela 5.1. Funkcije vezane za rukovanje greškama u CATCH bloku**

| Naziv funkcije | Opis |
|---|---|
| ERROR_LINE() | Broj linije koda u kojoj se pojavila greška |
| ERROR_NUMBER() | SQL Server broj greške |
| ERROR_MESSAGE() | Poruka o pojavi greške, koja sadrži sve vrednosti koje se prosleđuju u obliku parametara |
| ERROR_PROCEDURE() | Ukoliko se javi greška u proceduri, vraća se naziv procedure, u suprotonom, vraća se NULL vrednost |
| ERROR_SEVERITY() | Nivo ozbiljnosti greške |
| ERROR_STATE() | Vrednost stanja greške |

Ove sistemske funkcije mogu se koristiti za beleženje informacija o pojavi grešaka u tabelu, ili za prikazivanje poruka o pojavi grešaka korisniku. U sledećem primeru prikazuju se informacije o pojavi greške u rezultujućem skupu pomoću jedne vrste:

```
USE AdventureWorks2008;
BEGIN TRY
  SELECT 1/0;
END TRY
BEGIN CATCH
  INSERT INTO dbo.ErrorLog (Line, Number, ErrorMsg,
      [Procedure], Severity, [State])
  VALUES (ERROR_LINE(), ERROR_NUMBER(), ERROR_MESSAGE(),
      ERROR_PROCEDURE(), ERROR_SEVERITY(), ERROR_STATE());
END CATCH;
```

#### RUKOVANJE GREŠKAMA POMOĆU @@ERROR VREDNOSTI

Još jedan način za rukovanje greškama u toku izvršavanja korišćenjem Transact-SQL jezika je provera `@@ERROR` vrednosti nakon svake izvršene naredbe koja može potencijalno da generiše grešku. Ukoliko se pojavi greška, `@@ERROR` će sadržati broj greške. Ukoliko nema grešaka, vrednost će biti jednaka 0.

Korišćenje `@@ERROR` vrednosti dovodi do nešto drugačijeg stila kreiranja koda za rukovanje greškama, koji je najčešće mnogo teže održavati. Logika za rukovanje greškama mora da bude locirana nakon svake naredbe koja može da dovede do pojave grešaka. Veoma je važno da sačuvate `@@ERROR` vrednost pre nego što obavite testiranje, jer se ova vrednost resetuje nakon svake SQL naredbe.

```
SELECT 1/0;
PRINT @@ERROR;
```

U prethodnom primeru, prikazuje se vrednost 8143, zato što se javila greška zbog deljenja nulom. Međutim, razmotrimo sledeći primer:

```
SELECT 1/0;
IF @@ERROR <> 0
 PRINT @@ERROR;
```

Prilikom izvršavanja prethodnog koda, dobija se rezultat 0, zato što je `@@ERROR` vrednost resetovana nakon izvršavanja `IF` naredbe.

Zbog toga, veoma često ćete videti kod u kome se koristi sledeća logika:

```
DECLARE @SaveError int;
SELECT 1/0;
SET @SaveError = @@ERROR;
IF @SaveError <> 0
  PRINT @SaveError;
```

Još jedan nedostatak korišćenja `@@ERROR` vrednosti je da se ovom strategijom može dobiti samo broj greške, a ne poruka o pojavi greške. Poruka o pojavi greške se može dobiti pomoću sys.messages pogleda na osnovu ovog broja greške, ali bi tada svi parametri bili zauvek izgubljeni. Možda ste narušili određeno ograničenje, ali bez vrednosti parametara nećete moći da znate koje konkretno ograničenje ste narušili. Glavna prednost korišćenja ovog metoda jeste da je podržan mnogo duže od `TRY...CATCH` metoda u Transact-SQL jeziku.

**ZBOG ČEGA TREBA KORISTITI @@ERROR?**

Možda ste došli u iskušenje da koristite `TRY...CATCH` konstrukciju gde god je to moguće. Korišćenje `@@ERROR` vrednosti je veoma specifično; međutim, to je nešto za šta postoji podrška u svim prethodnim SQL Server verzijama. Ukoliko vaš kod mora da se izvršava na SQL Server verzijama koje prethode SQL Server 2005 verziji, morate da izbegavate korišćenje `TRY...CATCH` konstrukcije, odnosno umesto toga neophodno je da koristite `@@ERROR` vrednost za rukovanje greškama.

#### PRIKAZIVANJE PORUKA O POJAVAMA GREŠAKA

Postoje situacije u kojima je neophodno detektovanje grešaka i prikazivanje korisnički definisanih poruka o pojavama grešaka. Ovakve poruke mogu da se smeštaju kao korisnički definisane greške u sys.messages tabeli, a mogu biti i dinamički tekst. Ukoliko se koristi dinamički tekst, broj greške je uvek iznad 50000. Korisnički definisane greške se kreiraju na nivou instance i mogu da dovedu do konflikta ukoliko više aplikacija pokuša da detektuje grešku označenu istim brojem. Upotrebite `sp_addmessage` sistemsku snimljenu proceduru za dodavanje korisnički definisanih poruka o pojavama grešaka.

Sledeći kod koristi se za definisanje greške sa odgovarajućim parametrom:

```
EXEC sp_addmessage 50005, — Identifikator greške
10, — Nivo opasnosti
'Trenutni identifikator baze podataka je: %d, naziv baze podataka je: %s.';
```

`RAISERROR` funkcija se koristi za generisanje greške kojom može da se rukuje pozivanjem procedure ili Transact-SQL koda. `RAISERROR` naredba se koristi u sledećoj osnovnoj formi:

```
RAISERROR ( { id_poruke | string_poruke | @lokalna_promenljiva }
{ ,ozbiljnost ,stanje }
[ ,argument [ ,...n ] ] )
[ WITH opcija [ ,...n ] ]
```

Kao što je prikazano, `RAISERROR` funkcija kao prvi argument može da koristi numerički identifikator greške ili string kojim se opisuje greška. Ozbiljnost ukazuje na tip greške koja se pojavila. Ozbiljnost označena brojem 10 je informacionog karaktera (što znači da se ne radi o veoma ozbiljnim greškama). Više informacija o raspoloživim nivoima ozbiljnosti možete pronaći u SQL Server Books Online mrežno dostupnom servisu.

U sledećem primeru koristi se RAISERROR funkcija sa dva parametra i korisnički definisana greška, onakva kakva je prikazana u prethodnom primeru:

```
DECLARE @DBID int;
DECLARE @DBNAME nvarchar(128);
SET @DBID = DB_ID();
SET @DBNAME = DB_NAME();
RAISERROR (50005,
10, — Ozbiljnost.
1, — Stanje.
@DBID, — Prvi argument zamene.
@DBNAME); — Drugi argument zamene.
GO
```

RAISERROR funkcija se može primenjivati u sprezi sa korisnički definisanim porukama o pojavi grešaka, ili prosleđivanjem poruke u nju. Ukoliko se poruka koristi umesto broja greške, broj poruke je uvek veći od 50000.

```
RAISERROR ('Custom Message',
10, — Ozbiljnost
1); — Stanje
```

## Osnove obrade transakcija

Transakcije obezbeđuju da se modifikacija ili skup modifikacija obrađuje u potpunosti, a da se, u suprotnom, sve nepotpuno izvršene modifikacije ponište. One mogu biti blisko povezane sa rukovanjem greškama u bazi podataka i to na osnovu poslovnih pravila definisanih u organizaciji. U ovom odeljku, mi razmatramo podrazumevano funkcionisanje transakcija u SQL Server okruženju, a opisane su i eksplicitne transakcije. Detaljnije informacije o obradi transakcija i procesu koji se odvija iza scene možete pronaći u poglavlju 15, "Transakcije i strategije zaključavanja".

Svi mi želimo da naše baze podataka imaju mogućnost da održavaju integritet podataka, a transakcije su kritična komponenta u takvim sistemima. Transakcije imaju četiri osnovna svojstva, koja su opisana u tabeli 5.2.

**Tabela 5.2. Svojstva transakcija**

| Svojstvo | Opis |
|---|---|
| Atomičnost | Svaka transakcija tretira se kao atomična operacija: ona se ili izvršava ili poništava u potpunosti. |
| Konzistentnost | Sve strukture baza podataka moraju da budu konzistentne. Tabele i indeksi moraju da ostanu sinhronizovani. |
| Izolacija | Transakcije su izolovane jedne od drugih pomoću brava. Nivo izolacije je utvrđen nivoom izolovanja same transakcije. Zaključavanje i nivoi izolovanja transakcija veoma detaljno su opisani u poglavlju 15. |
| Trajnost | Transakcije moraju da budu perzistentne u situacijama u kojima dolazi do otkaza sistema. Beleženje informacija o transakcijama u SQL Server okruženju je takvo da je prethodno navedeni zahtev ispunjen. |

### TRANSAKCIJE SA AUTOMATSKIM POTVRĐIVANJEM IZVRŠAVANJA

Na osnovu podrazumevanih podešavanja, SQL Server razmatra svaku naredbu koja izvršava neku modifikaciju kao da se radi o posebnoj transakciji. Ili će se naredba izvršiti u potpunosti, ili se neće izvršiti nikakvo modifikovanje baze podataka. Na primer, `UPDATE` naredba će ili izvršiti neophodno ažuriranje zapisa na osnovu navedenih kriterijuma za pretraživanje, ili neće izvršiti modifikovanje nijednog zapisa. Ukoliko se pojavi odgovarajuća greška u toku procesa modifikovanja zapisa, sve promene koje su izvršene nad blokovima moraće da budu poništene.

Veoma je važno da razumete funkcionisanje niza naredbi u Autocommit režimu. Ukoliko se pojavi odgovarajuća greška prilikom izvršavanja, naredba koja je generisala grešku u toku svog izvršavanja se poništava; međutim, niz naredbi nastavlja da se izvršava. Razmotrimo sledeći niz naredbi:

```
USE SandboxDB;
CREATE TABLE T3
 (c int PRIMARY KEY NOT NULL);
INSERT INTO T3 VALUES (1);
INSERT INTO T3 VALUES (2/0);
INSERT INTO T3 VALUES (3);
SELECT * FROM T3;
```

Prilikom izvršavanja ovog niza naredbi, kreira se odgovarajuća tabela, a prva `INSERT` naredba se uspešno izvršava. Druga `INSERT` naredba u ovom nizu naredbi uzrokuje pojavu greške u toku izvršavanja, jer dolazi do deljenja sa nulom. Druga `INSERT` naredba se terminira usled pojave greške prilikom izvršavanja, ali se zato nastavlja sa izvršavanjem sledeće `INSERT` naredbe. Prilikom analiziranja rezultata izvršavanja ovog niza naredbi pomoću `SELECT` naredbe, tabela T3 će sadržati samo dve vrste.

Međutim, ukoliko se pojavi sintaksna greška, celokupan niz naredbi se neće izvršiti. Razmotrimo sledeći niz naredbi:

```
USE SandboxDB;
CREATE TABLE T4
 (c int PRIMARY KEY NOT NULL);
INSERT INTO T4 VALUES (1);
INSERT INTO T4 VALUES (2//0);
INSERT INTO T4 VALUES (3);
SELECT * FROM T4;
```

U prethodnom primeru, postoji sintaksna greška u drugoj `INSERT` naredbi. Zbog pojave ove greške, niz naredbi ne može da se prevede u odgovarajući plan izvršavanja, tako da se ne izvršava nijedna naredba, uključujući tu i `CREATE TABLE` naredbu.

### EKSPLICITNE TRANSAKCIJE

SQL Server vam omogućava da grupu naredbi postavite u jednu transakciju. `BEGIN TRANSACTION` naredba ukazuje na početak transakcije. `BEGIN TRANSACTION` naredba ukazuje da sve promene koje su izvršene od `BEGIN TRANSACTION` naredbe treba da budu potvrđene u bazi podataka. `ROLLBACK TRANSACTION` naredba omogućava da sve modifikacije izvršene nad bazom podataka poništi sistem za upravljanje bazom podataka.

`XACT_ABORT` je opcija vezana za sesiju, kojom se kontroliše ponašanje grešaka u toku izvršavanja transakcija. `XACT_ABORT` opcija može da ima vrednost `ON` ili `OFF`, a podrazumevana vrednost opcije je `OFF`. Ukoliko ova opcija ima vrednost `ON`, sve naredbe koje uzrokuju pojavu greške automatski će izazvati poništavanje transakcije i sprečiti dalje izvršavanje niza naredbi. Razmotrimo sledeći primer:

```
USE SandboxDB;
SET XACT_ABORT ON;
CREATE TABLE T5
 (c int PRIMARY KEY NOT NULL);
BEGIN TRAN;
  INSERT INTO T5 VALUES (1);
  INSERT INTO T5 VALUES (2);
  INSERT INTO T5 VALUES (2);
  INSERT INTO T5 VALUES (3);
COMMIT TRAN;
SELECT * FROM T5;
```

Narušavanje ograničenja vezanih za primarni ključ tabele javlja se u trećoj `INSERT` naredbi u prethodno prikazanoj transakciji. Pošto `XACT_ABORT` opcija ima vrednost `ON`, pojava ove greške će automatski izazvati poništavanje transakcije, odnosno sprečavanje daljeg izvršavanja niza naredbi. `SELECT` naredba iz ovog primera nikada se neće izvršiti.

Ukoliko bi se isti skript izvršavao kada `XACT_ABORT` opcija ima vrednost `OFF`, greška prilikom izvršavanja izazvala bi poništavanje samo tekuće naredbe koja je izazvala tu grešku, a ostale naredbe u nizu bi se izvršavale. Rezultat izvršavanja `SELECT` naredbe su tri vrste sa podacima.

Pošto `XACT_ABORT` opcija ima inicijalnu vrednost `OFF`, kritični delovi transakcije treba da budu postavljeni u `TRY...CATCH` blokovima, kao što je prikazano u sledećem primeru:

```
USE SandboxDB;
CREATE TABLE T6
 (c int PRIMARY KEY NOT NULL);
BEGIN TRY
  BEGIN TRAN;
    INSERT INTO T6 VALUES (1);
    INSERT INTO T6 VALUES (2);
    INSERT INTO T6 VALUES (2);
    INSERT INTO T6 VALUES (3);
  COMMIT TRAN;
END TRY
BEGIN CATCH
  PRINT 'An Error Occured';
  ROLLBACK TRAN;
END CATCH;
SELECT * FROM T6;
```

## Snimljene procedure

Snimljene procedure su skupovi operacija, koji su snimljeni na serveru i koje izvršavaju klijentske aplikacije. Vrednosti parametara se mogu prosleđivati u snimljene procedure. Izlazni parametri se mogu koristiti kao vrednosti promenljivih u kodu iz koga se pozivaju snimljene procedure. Snimljene procedure mogu se definisati sa maksimalno 2100 parametara. Ukoliko se kao rezultat vraća samo jedna celobrojna vrednost, to je obično indikator uspešnosti izvršavanja procedure.

Postoji veliki broj operacija koje se mogu izvršavati putem procedura snimljenih u bazama podataka. Modifikovanje strukture baze podataka i izvršavanje korisnički definisanih transakcija su najčešće operacije. Snimljene procedure se mogu primenjivati za vraćanje rezultata izvršavanja `SELECT` naredbi, ali u opštem slučaju su mnogo manje fleksibilne, kada se radi o rezultatima, u odnosu na korisnički

definisane funkcije koje vrednosti izračunavanja smeštaju u odgovarajuće tabele. Kasnije u toku ovog poglavlja biće detaljnije opisane korisnički definisane funkcije.

U ovom odeljku fokusiraćemo se na kreiranje Transact-SQL snimljenih procedura, ali je moguće da u bazu podataka snimite procedure i korišćenjem .NET integracije. Prilikom kreiranja CLR snimljenih procedura, možete koristiti bilo koji .NET jezik, uključujući C# i VB.NET. O tome će više reči biti u poglavlju 18, "SQL Server .NET CLR".

Velika je prednost korišćenja snimljenih procedura, od bezbednosti, modularnosti prilikom programiranja, pa do redukovanja ukupnog mrežnog saobraćaja.

**Bezbednost** Snimljene procedure garantuju pristupanje nezavisno od objekata baze podatka koje one referenciraju. Uopšteno rečeno, korisnik kome je omogućeno izvršavanje snimljene procedure ima mogućnost da izvršava sve operacije koje su definisane tom procedurom. Postoji mogućnost da snimljenu proceduru izvršavate kao neki drugi korisnik. Više informacija o pristupnim privilegijama i dozvolama, odnosno izvršavanju koje je identično sa izvršavanjem nekog drugog korisnika, možete naći u poglavlju 8, "Upravljanje bezbednošću korisnika".

**Modularno programiranje** Snimljene procedure promovišu višestruko korišćenje Transact-SQL logike u aplikacijama. One omogućavaju da se složenije procedure razdele na nekoliko manjih modula.

**Redukovanje mrežnog saobraćaja** Korišćenjem snimljenih procedura možete redukovati količinu podataka koji se razmenjuju mrežnim putem između klijenta i servera. Slanje poziva za izvršavanje snimljene procedure, koja sadrži više od 100 linija koda, zapravo, zahteva slanje mnogo manje podataka od slanja 100 linija koda svaki put kada je neophodno da se obavi odgovarajuća obrada. Takođe, smanjuje se količina podataka koje je neophodno razmenjivati prilikom izvršavanja višestrukih operacija nad bazom podataka.

## Projektovanje efikasnih snimljenih procedura

U prethodnim SQL Server verzijama snimljene procedure su predstavljale jedini način za izvršavanje odgovarajućih operacija. Veoma je važno da u potpunosti razumete opcije koje su raspoložive prilikom rešavanja problema, umesto da odmah izaberete kreiranje procedure za svaku operaciju koju ćete izvršavati nad bazom podataka. Snimljene procedure mogu da sadrže skoro svaku Transact-SQL naredbu, uz nekoliko izuzetaka.

Objekat, kao što je tabela, može se kreirati i koristiti u istoj proceduri, pod uslovom da se prvo kreira odgovarajuća tabela.

Sledeće operacije ne smeju da se nalaze u procedurama:

- Kreiranje ili modifikovanje sledećih objekata:
  - Agregatne funkcije
  - Podrazumevane vrednosti
  - Funkcije
  - Procedure
  - Pravila
  - Šeme
  - Trigera
  - Pogleda
- USE naredba
- `SET PARSEONLY ili SHOWPLAN` varijante

Objekti koji se ne nalaze u prethodnoj listi mogu se kreirati, modifikovati ili uklanjati u trenutno korišćenoj bazi podataka u okviru snimljene procedure. Modifikacije baza podataka i korisnički definisane transakcije su primarni kandidati za enkapsulaciju u obliku snimljene procedure.

Snimljena procedura može da vrati jedan ili više rezultujućih skupova u aplikaciju iz koje se izvršava procedura. Korisnički definisana funkcija, koja vrednosti smešta u tabelu, treba da se koristi umesto snimljene procedure kada je neophodno vratiti samo jedan rezultujući skup podataka. Rezultati iz snimljene procedure se ne mogu koristiti u `FROM` klauzuli upita.

U snimljenim procedurama možete da koristite privremene tabele. Svaka privremena tabela traje onoliko koliko traje i izvršavanje snimljene procedure. Ugnežđena snimljena procedura može da pristupa lokalnim privremenim tabelama koje su kreirane u snimljenoj proceduri iz koje je ta snimljena procedura pozvana.

Prilikom referenciranja objekata unutar snimljene procedure, oni treba da budu definisani uz navođenje naziva šeme. Ukoliko to ne uradite, prvo podrazumevano pretraživanje se obavlja u okviru šeme u kojoj je definisana snimljena procedura. Ukoliko odgovarajući objekat nije pronađen, pretražuje se korisnička podrazumevana šema. Ovo može dovesti do pojave grešaka za korisnika koji nema definisanu odgovarajuću podrazumevanu šemu, ili je korisnička podrazumevana šema izmenjena.

Prilikom kreiranja procedure koja se snima na serveru, snimaju se dve opcije sesije. To su `QUOTED_IDENTIFIERS` i `ANSI_NULLS` opcije. `QUOTED_IDENTIFIERS` opcijom se utvrđuje da li se dvostruki navodnici koriste za označavanje identifikatora, ili treba da se interpretiraju kao string literali. `ANSI_NULLS` opcijom se utvrđuje kako se obavlja poređenje u odnosu na `NULL` vrednosti. Ukoliko je izabrana `ANSI_NULLS` opcija, jedino će poređenje sa `NULL` vrednostima korišćenjem IS operatora vratiti vrste u `SELECT` naredbu.

Nakon što je kreirana, snimljena procedura može da referencira tabele koje u određenom trenutku ne postoje. Nazivi tabela se analiziraju prilikom prvog izvršavanja procedure. Ukoliko tabela postoji prilikom kreiranja snimljene procedure, sve referencirane kolone moraju da postoje u vreme kreiranja.

## Kreiranje i izvršavanje jednostavne snimljene procedure

Počnimo od kreiranja veoma jednostavne procedure. Sledeća procedura koju ćete snimiti na serveru projektovana je tako da se izvršava u okviru `CATCH` bloka `TRY…CATCH` konstrukcije:

```
USE SandboxDB;
GO
CREATE PROC dbo.uspPrintError
AS
PRINT 'Error ' + CONVERT(varchar(50), ERROR_NUMBER()) +
  ', Severity ' + CONVERT(varchar(5), ERROR_SEVERITY()) +
  ', State ' + CONVERT(varchar(5), ERROR_STATE()) +
  ', Procedure ' + ISNULL(ERROR_PROCEDURE(), '-') +
  ', Line ' + CONVERT(varchar(5), ERROR_LINE());
PRINT ERROR_MESSAGE();
GO
```

U ovoj proceduri, prikazuju se informacije o identifikovanim greškama. Ovaj kod se može koristiti u bilo kojoj situaciji u kojoj želite da prikazujete informacije o greškama, i time se promoviše višestruko korišćenje koda.

Pokušajte sada da izvršite ovu proceduru. Zapamtite, pošto ova procedura koristi funkcije za očitavanje podataka o greškama, neophodno je da je pozivate u okviru CATCH bloka. Da biste pregledali dobijene poruke, neophodno je da pređete na Messages karticu.

```
BEGIN TRY
 SELECT 1/0;
END TRY
BEGIN CATCH
 EXEC dbo.uspPrintError;
END CATCH;
```

Izvršavanjem prethodno navedenih Transact-SQL naredbi dobija se sledeći rezultat:

```
Error 8134, Severity 16, State 1, Procedure -, Line 2
Divide by zero error encountered.
```

Sadržaj snimljene procedure može se modifikovati korišćenjem `ALTER` naredbe. Korišćenjem `ALTER` naredbe se, zapravo, menja definicija procedure, pri čemu se ne utiče na promenu privilegija vezanih za posmatranu proceduru. U sledećem kodu vrši se modifikovanje `uspPrintError` snimljene procedure, kako bi se prikazivale poruke o korisnički definisanim greškama ukoliko je broj greške jednak `NULL`:

```
USE SandboxDB;
GO
ALTER PROC dbo.uspPrintError
AS
IF ERROR_NUMBER() IS NULL
 BEGIN
 PRINT 'This procedure must be used within a CATCH block';
 RETURN(1);
 END
ELSE
 BEGIN
 PRINT 'Error ' + CONVERT(varchar(50), ERROR_NUMBER()) +
   ', Severity ' + CONVERT(varchar(5), ERROR_SEVERITY()) +
   ', State ' + CONVERT(varchar(5), ERROR_STATE()) +
   ', Procedure ' + ISNULL(ERROR_PROCEDURE(), '-') +
   ', Line ' + CONVERT(varchar(5), ERROR_LINE());
 PRINT ERROR_MESSAGE();
 END
GO
```

Ukoliko neka od prethodno snimljenih procedura nije više neophodna u posmatranoj bazi podataka, ona se uklanja na sledeći način:

```
USE SandboxDB;
DROP PROC dbo.uspPrintError;
```

## Upotreba parametara

U najvećem broj snimljenih procedura koriste se odgovarajući parametri. Parametri se mogu koristiti za obezbeđivanje ulaznih vrednosti neophodnih za izvršavanje procedure, kao i za prosleđivanje rezultata izvršavanja procedure do koda iz koga se procedura poziva. U ovom odeljku ćemo razmotriti nekoliko primera korišćenja različitih tipova parametara.

### Ulazni parametri

Ulazni parametri su promenljive koje su definisane u zaglavlju snimljene proedure. Prilikom izvršavanja snimljene procedure, vrednosti ovih promenljivih moraju da budu na raspolaganju. U

sledećem primeru prikazana je procedura u kojoj su navedena tri načina za definisanje ulaznih parametara:

```
USE SandboxDB;
GO
CREATE PROC uspInputParam
 @param1 int = 5,
 @param2 int = NULL,
 @param3 int
AS
IF @param2 IS NULL
BEGIN
 PRINT 'You must supply a value for @param2.';
 RETURN(1);
END
PRINT @param1 + @param2 + @param3;
GO
```

U prethodnoj proceduri, u zaglavlju su definisana tri ulazna parametra. `@param1` i `@param2` imaju odgovarajuće podrazumevane vrednosti. Za drugi parametar proverava se da li ima vrednost `NULL` pre nego što se omogući dalje izvršavanje procedure. Ukoliko je vrednost i dalje jednaka `NULL`, prikazuje se prigodna poruka o pojavi greške. `RETURN` naredba omogućava bezuslovno napuštanje procedure. Treći parametar ne sadrži podrazumevanu vrednost i SQL Server ga smatra obaveznim parametrom. Prva dva parametra se smatraju opcionim parametrima, pošto je za njih obezbeđena podrazumevana vrednost, koja se koristi ukoliko vrednosti ovih parametara nisu navedene.

Razmotrimo različite metode za izvršavanje ove procedure korišćenjem Transact-SQL jezika. Jedna od opcija je prosleđivanje parametara u redosledu u kome su oni definisani u zaglavlju snimljene procedure:

```
— prosleđivanje vrednosti parametara u definisanom redosledu
EXEC uspInputParam 1, 2, 3;
```

Prilikom izvršavanja koda, prikazuje se broj 6, pošto smo obezbedili vrednosti za sve neophodne parametre.

```
— prosleđivanje vrednosti parametara na osnovu pozicije, bez navođenja svih vrednosti
EXEC uspInputParam 1, 2;
```

Ukoliko izostavite bilo koji od neophodnih parametara, SQL Server generiše poruku o pojavi greške:

```
Msg 201, Level 16, State 4, Procedure uspInputParam, Line 0
Procedura ili funkcija 'uspInputParam' očekuje parametar '@param3', koji nije naveden.
```

Prosleđivanje parametara moguće je i na osnovu naziva. Prilikom prosleđivanja parametara na osnovu naziva, možete da preskočite neki od parametara, odnosno možete da navodite vrednosti parametara u proizvoljnom redosledu.

```
— prosleđivanje vrednosti parametara na osnovu naziva; izostavljena je vrednost
— parametra @param2
EXEC uspInputParam @param3 = 5;
```

U prethodnom primeru smo obezbedili vrednost samo trećeg parametra. Pošto se `IF` naredba nalazi na početku procedure, prikazuje se sledeća poruka o pojavi greške:

```
You must supply a value for @param2.
```

U poslednjem izvršavanju ove snimljene procedure koristićemo podrazumevanu vrednost prvog parametra:

```
— prosleđivanje vrednosti parametara na osnovu naziva; koristi se podrazumevana
— vrednost parametra @param1
EXEC uspInputParam @param3 = 5,
@param2 = 5;
```

Zbog toga što su sve neophodne vrednosti obezbeđene, snimljena procedura se izvršava, a rezultat izvršavanja procedure je 15.

Još jedan, mnogo realniji primer korišćenja ulaznih parametara je prikazan u narednom primeru. U ovom primeru se koristi AdventureWorks2008 baza podataka i ažurira se informacija o zaradi zaposlenog, odnosno beleže se sve promene u EmployeePayHistory tabeli.

```
USE AdventureWorks2008;
GO
CREATE PROCEDURE HumanResources.uspUpdateEmployeeHireInfo
 @BusinessEntityID int,
 @JobTitle nvarchar(50),
 @HireDate datetime,
 @RateChangeDate datetime,
 @Rate money,
 @PayFrequency tinyint,
 @CurrentFlag dbo.Flag
AS
SET NOCOUNT ON;
BEGIN TRY
 BEGIN TRANSACTION;
 UPDATE HumanResources.Employee
 SET JobTitle = @JobTitle
  ,HireDate = @HireDate
  ,CurrentFlag = @CurrentFlag
 WHERE BusinessEntityID = @BusinessEntityID;
 INSERT INTO HumanResources.EmployeePayHistory
  (BusinessEntityID
  ,RateChangeDate
  ,Rate
  ,PayFrequency)
 VALUES (@BusinessEntityID, @RateChangeDate, @Rate, @PayFrequency);
 COMMIT TRANSACTION;
END TRY
BEGIN CATCH
 IF @@TRANCOUNT > 0
 BEGIN
  ROLLBACK TRANSACTION;
 END
 EXECUTE dbo.uspLogError;
END CATCH;
```

Ova snimljena procedura koristi se za izvršavanje modifikacija u korisnički definisanoj transakciji. Ukoliko se javi bilo kakva greška, transakcija se poništava, a informacije o greškama koje su se pojavile prilikom izvršavanja transakcije se beleže u odgovarajućoj datoteci izvršavanjem snimljene procedure.

#### IZLAZNI PARAMETRI

Izlazni parametri vam omogućavaju da prosleđujete vrednosti podataka iz snimljene procedure korišćenjem odgovarajućih promenljivih. Navođenje ključne reči `OUTPUT` je neophodno i prilikom definisanja parametra, ali i prilikom referenciranja parametra u toku izvršenja.

Prilikom vraćanja jednog zapisa iz snimljene procedure, korišćenje nekoliko izlaznih parametara može da obezbedi mnogo efikasnije izvršavanje od vraćanja klijentu rezultujućeg skupa koji sadrži samo jedan zapis.

U sledećem primeru očitava se AccountNumber vrednost za konkretnog kupca:

```
USE AdventureWorks2008;
GO
CREATE PROC Sales.uspGetCustomerAccountNumber
  @CustomerID int,
  @AccountNumber varchar(10) OUTPUT
AS
SELECT @AccountNumber = AccountNumber
 FROM Sales.Customer
 WHERE CustomerID = @CustomerID;
GO
```

Kako bismo izvršavali ovu proceduru korišćenjem Transact-SQL jezika, neophodno je da obezbedimo promenljivu za smeštanje naših izlaznih parametara, pri čemu se ključna reč `OUTPUT` navodi iza naziva odgovarajuće promenljive. Obratite pažnju na to da naziv promenljive u kojoj se nalazi izlazna vrednost i naziv parametra ne moraju da budu identični. Primer izvršavanja je prikazan u sledećem kodu:

```
USE AdventureWorks2008;
DECLARE @Acct varchar(10);
EXEC Sales.uspGetCustomerAccountNumber 1, @Acct OUTPUT;
PRINT @Acct;
```

Još jedan primer korišćenja izlaznih parametara postoji u `uspLogError` snimljenoj proceduri, koju smo imali priliku da prethodno isprobamo. Obratite pažnju na to da čak i ako obezbedimo izlazni parametar, on ne mora da bude vraćen prilikom izvršavanja snimljene procedure.

```
USE AdventureWorks2008;
GO
ALTER PROCEDURE dbo.uspLogError
 @ErrorLogID int = 0 OUTPUT
AS
SET NOCOUNT ON;
BEGIN TRY
 -- Vraća se vrednost ukoliko ne postoji informacija o pojavi grešaka
 IF ERROR_NUMBER() IS NULL
  RETURN 1;
 -- Vraća se vrednost ukoliko se izvršava transakcija koja ne može da se potvrdi.
 -- Unos/modifikovanje podataka nije dozvoljeno kada se
 -- transakcija nalazi u stanju u kome nije moguće potvrđivanje njenog izvršavanja.
 IF XACT_STATE() = -1
 BEGIN
  PRINT 'Cannot log error since the current transaction ' +
```

```
        'is in an uncommittable state. '
  RETURN 1;
 END
 INSERT dbo.ErrorLog
  (UserName, ErrorNumber, ErrorSeverity,
  ErrorState, ErrorProcedure, ErrorLine, ErrorMessage)
 VALUES
  (CONVERT(sysname, CURRENT_USER), ERROR_NUMBER(), ERROR_SEVERITY(),
  ERROR_STATE(), ERROR_PROCEDURE(), ERROR_LINE(), ERROR_MESSAGE());
-- Prosleđivanje ErrorLogID vrednosti unete vrste
 SET @ErrorLogID = @@IDENTITY;
END TRY
BEGIN CATCH
 PRINT 'An error occurred in stored procedure uspLogError: ';
 EXECUTE dbo.uspPrintError;
 RETURN -1;
END CATCH
```

## Upravljanje povratnim vrednostima

Vrednosti koje vraća procedura treba da se koriste i u cilju provere uspešnosti izvršavanja same snimljene procedure. Ukoliko drugačije nije definisano, sistemske snimljene procedure prilikom neuspešnog izvršavanja vraćaju vrednost koja je različita od nule.

Da biste vratili bilo koju drugu informaciju iz snimljene procedure, neophodno je da se koriste izlazni parametri. Snimljena procedura može da vrati jednu celobrojnu vrednost, ali zato može da sadrži proizvoljan broj izlaznih parametara.

Prilikom izvršavanja snimljene procedure, kada se dođe do `RETURN` naredbe, prekida se izvršavanje procedure i vraća na kod iz koga je ta procedura pozvana. Podrazumevana povratna vrednost je 0, što u suštini podrazumeva uspešno izvršavanje. Vrednosti različite od nule obično ukazuju na postojanje odgovarajuće greške. `NULL` vrednost nikada neće biti vraćena iz snimljene procedure; biće konvertovana u vrednost 0, odnosno generisaće se odgovarajuće upozorenje.

Da biste detektovali povratnu vrednost snimljene procedure, neophodno je da primenite sledeću sintaksu:

```
CREATE PROC uspMyProc
AS
RETURN 1;
GO
DECLARE @retval int;
EXEC @retval = uspMyProc;
PRINT @retval;
```

Korisnički definisane funkcije nude mnogo veću fleksibilnost kada se radi o povratnim vrednostima, a one će biti detaljnije razmatrane kasnije u toku ovog poglavlja.

## Razumevanje kompilacije

Pošto smo opisali način na koji se kreiraju i pozivaju procedure koje se čuvaju na serveru baza podataka, neophodno je da razmotrimo i kako se one prevode i izvršavaju.

Prilikom kreiranja nove procedure koja se čuva na serveru, ona se analizira kako bi se proverila njena sintaksna ispravnost, a zatim se skladište Transact-SQL naredbe. Naredbe unutar snimljene procedure se ne prevode u plan izvršenja sve dok se procedura ne pozove prvi put. Neophodno je odgovarajuće analiziranje naziva – što je dobro, jer objekti mogu da referenciraju nešto što nije definisano.

Prilikom prvog izvršavanja odgovarajućeg upita, kreira se plan izvršenja na osnovu naredbi i navedenih vrednosti parametara. Plan izvršenja se, zatim, postavlja u keš za izvršavanje procedura.

Plan izvršenja sadrži dve komponente: plan upita i kontekst izvršavanja. Plan upita definiše fizički proces i redosled u kome se javljaju operacije vezane za bazu podataka. Za datu Transact-SQL naredbu, može da postoji nekoliko načina za izvršavanje fizičkih operacija. Funkcija optimizatora upita je da kreira efikasan plan upita na osnovu korišćenih naredbi i raspoloživih vrednosti. Optimizator upita utvrđuje indekse koje je neophodno koristiti, redosled izvršavanja operacija, strategiju povezivanja tabela i mnoge druge operacije kako bi se dobio efikasan plan izvršenja. Optimizator upita je detaljno opisan u poglavlju 14, "Strategije indeksiranja za optimizovanje upita".

Najveći broj SQL naredbi se može podeliti na komponente izvršenja i vrednosti parametara. Kontekst izvršenja sastoji se od vrednosti parametara za svako pojedinačno izvršenje snimljene procedure. Plan upita se može primenjivati više puta, za različite vrednosti parametara. Plan upita i kontekst izvršenja predstavljaju plan izvršenja.

U određenom trenutku planovi izvršenja moraju da budu uklonjeni iz keša vezanog za izvršenje procedura. Planovi koji nisu primenjivani određeni vremenski period su zastareli, tako da je neophodno da budu uklonjeni iz keša, kako bi se napravio prostor za nove planove izvršenja. Strukturne promene baze podataka, uključujući i promene strukture referenciranih objekata ili indeksa, mogu takođe da promene planove izvršenja.

Ukoliko je uklonjen indeks koji koristi plan izvršenja, plan mora da se ponovo generiše, kako bi mogao da koristi raspoložive objekte. Ovo automatski detektuje i realizuje okruženje za upravljanje bazama podataka.

SQL Server 2008 obezbeđuje rekompilaciju na nivou naredbi za snimljene procedure. Kada se u bazi podataka identifikuje promena koja zahteva ponovno kreiranje plana, kompiliraju se samo one naredbe na koje utiču promene.

#### UPRAVLJANJE KOMPILACIJOM

Postoji nekoliko situacija u kojima ćete želeti da upravljate procesom prevođenja vezanim za planove izvršavanja.

Ponekad snimljene procedure prihvataju veoma veliki opseg vrednosti parametara. Najčešća situacija je pretraživanje opsega. Ukoliko upit vraća pet zapisa, verovatno će biti sasvim drugačiji plan ukoliko se dobija pet miliona zapisa. Snimljena procedura koja dobija veoma različite opsege vrednosti može da ima korist od rekompajliranja prilikom svakog izvršenja. To se može realizovati kreiranjem procedure uz navođenje `WITH RECOMPILE` opcije, kao što je prikazano u sledećem primeru:

```
USE AdventureWorks2008;
GO
CREATE PROCEDURE uspOrderRange
 @BeginDate datetime,
 @EndDate datetime
WITH RECOMPILE
AS
SELECT *
 FROM Sales.SalesOrderHeader
 WHERE OrderDate BETWEEN @BeginDate AND @EndDate;
```

`WITH RECOMPILE` opcija može da se navede i prilikom izvršavanja snimljene procedure. Korišćenjem ove sintakse omogućava se rekompajliranje snimljene procedure za samo jedno izvršenje. Za sledeća izvršavanja će se koristiti keširani plan:

```
EXEC uspOrderRange '12/1/2006', '12/1/2009' WITH RECOMPILE;
```

Planovi izvršenja se automatski poništavaju onda kada je to neophodno, ali ne postoji mogućnost da koriste objekte koji su kreirani nakon što je definisan plan izvršenja. Na primer, ukoliko je plan izvršenja u kešu, kreira se indeks koji može biti od koristi, SQL Server će nastaviti da koristi keširani plan sve dok on ne bude uklonjen iz keša. Naredba `sp_recompile` se može koristiti za označavanje planova koje je neophodno ponovo kreirati:

- Individualne snimljene procedure
- Individualne korisnički definisane funkcije
- Tabele
- Pogledi

Nakon kreiranja novih indeksa za tebele, dobra ideja je da izvršite sp_recompile naredbu, kako biste označili sve procedure koje koriste posmatranu tabelu, jer je neophodno ponovno prevođenje.

# Pogledi

*Pogled* je upit koji je smešten na serveru kao objekat i referencira se kao tabela. Sa izuzetkom indeksiranih pogleda, rezultati upita nisu snimljeni u okviru baze podataka, već samo definicija upita.

Korišćenje pogleda je veoma slično korišćenju tabele. Rezultati pogleda se mogu spajati sa drugim tabelama u `FROM` klauzuli upita. Dobra ideja je da koristite prefiks za identifikovanje pogleda prilikom izbora naziva, tako da se može veoma jasno razlikovati od tabele.

Pogled može da referencira neke druge poglede. Ova funkcionalnost može da dovede do problema kada se radi o performansama, jer SQL Server mora da kombinuje komponente nekoliko `SELECT` naredbi u jednom planu izvršenja. Ukoliko je to moguće, treba da pokušate da direktno referencirate tabele. Ukoliko je neophodno da jedan pogled referencira neki drugi pogled, proverite da li je kojim slučajem nepotrebno upotrebljena dodatna logika.

Pogled se može modifikovati navođenjem nekih ograničenja. `INSERT`, `UPDATE` i `DELETE` naredbe su omogućene, ukoliko su ispunjeni sledeći uslovi:

- U jednom trenutku mogu da se izvršavaju promene samo jedne tabele.
- Modifikacije moraju da poštuju sva ograničenja odgovarajuće tabele.
  - Ukoliko pogled ne obezbeđuje sve kolone bez podrazumevanih vrednosti, ne možete da definišete pogled.
- Ne mogu se modifikovati kolone u kojima se nalaze izračunate vrednosti, odnosno pogledi u kojima se korsite `GROUP BY` operacije.

Postoji mogućnost da omogućite modifikacije pogleda pisanjem neophodne logike u `INSTEAD OF` trigeru. O tome će biti nešto više reči u poglavlju 6, "Upravljanje integritetom podataka".

## Zbog čega je potrebno koristiti poglede?

Pogledi mogu biti od velike koristi programerima baza podataka. Prednosti korišćenja pogleda prilikom projektovanja baza podataka su:

**Pogledi pojednostavljuju pribavljanje podataka.** Standardni upiti mogu se definisati u obliku pogleda u bazi podataka, kako bi se pojednostavilo kreiranje upita. Izračunavanja, izvedene kolone i logika za grupisanje mogu da omoguće proces kreiranja upita mnogo jednostavnijim.

**Pogledima se korisnici fokusiraju na specifične podatke.** Pogled omogućava korisniku da se fokusira samo na relevantne podatke, tako što filtrira sve nepotrebne ili osetljive kolone i vrste. Pogled može da referencira sistemske funkcije u cilju daljeg fokusiranja podataka.

**Pogledi obezbeđuju apstrakciju promena.** Kada aplikacija referencira poglede i snimljene procedure, ona može da omogući programeru baza podataka da mnogo fleksibilnije izvrši kasnije promene u strukturi tabela. Kako se menja struktura tabela, pogledi mogu da se modifikuju tako da kompenzuju neophodne izmene aplikacija koje pristupaju podacima.

**Pogledi obezbeđuju bolje performanse.** Indeksirani pogled može da obezbedi bolje performanse smeštanjem rezultujućeg skupa u bazu podataka. Za operacije koje se izvršavaju nad bazom podataka, a koje grupišu podatke, to može da obezbedi značajno poboljšanje kada se radi o performansama. Indeksirani pogledi su detaljnije obrađeni u Poglavlju 14.

**Pogledi izdvajaju bezbednosne objekte.** Pogled može definisati privilegije nezavisno od tabele. To znači da neko može da ima pristup pogledu bez potrebe da pristupa određenoj tabeli, sve dok je vlasnik oba objekta isti korisnik baze podataka.

SQL Server proverava privilegije svaki put kada neki objekat referencira neki drugi objekat čiji je vlasnik drugi korisnik, a to je uslov koji može da rezultuje prekidom lanca vlasništva.

## Standardni pogledi

Kreiranje pogleda je veoma jednostavan zadatak. Glavni korak je kreiranje upita koji generiše željeni rezultujući skup podataka. U sledećem primeru očitava se naziv i titula svakog zaposlenog čiji se podaci nalaze u AdventureWorks2008 bazi podataka:

```
USE AdventureWorks2008;
GO
CREATE VIEW HumanResources.vEmployeeInfo
AS
SELECT p.Firstname,
       p.LastName,
       e.JobTitle
 FROM Person.Person p
 INNER JOIN HumanResources.Employee e
 ON p.BusinessEntityID = e.BusinessEntityID
WHERE e.CurrentFlag = 1;
GO
```

Nakon definisanja pogleda, on može da se koristi na isti način kao što biste koristili tabelu, što je prikazano u sledećem primeru.

```
SELECT * FROM HumanResources.vEmployeeInfo
 WHERE JobTitle LIKE '%manager%';
```

Izmena pogleda je identična izmeni snimljene procedure. Definicija pogleda se menja, ali sve privilegije vezane za pogled i dalje važe. U sledećem primeru u `vEmployeeInfo` pogled dodaje se polje vezano za odeljenje:

```
ALTER VIEW HumanResources.vEmployeeInfo
AS
SELECT p.Firstname,
       p.LastName,
       e.JobTitle,
       d.Name As DepartmentName
 FROM Person.Person p
 INNER JOIN HumanResources.Employee e
 ON p.BusinessEntityID = e.BusinessEntityID
 INNER JOIN HumanResources.EmployeeDepartmentHistory edh
 ON edh.BusinessEntityID = e.BusinessEntityID
 INNER JOIN HumanResources.Department d
 ON d.DepartmentID = edh.DepartmentID
WHERE e.CurrentFlag = 1 AND
  edh.EndDate IS NULL;
```

Pogled može da se preimenuje korišćenjem `sp_rename` sistemske snimljene procedure. Prilikom preimenovanja pogleda, originalni naziv ostaje u samoj definiciji pogleda.

Da biste uklonili pogled, neophodno je da upotrebite `DROP VIEW` naredbu, kao što je prikazano u sledećem primeru:

```
DROP VIEW HumanResources.vEmployeeInfo;
```

#### Opcije vezane za poglede

Prilikom kreiranja pogleda možete da navedete sledeće opcije: `WITH SCHEMABINDING` i `WITH CHECK OPTION`.

Kreiranje pogleda navođenjem `WITH SCHEMABINDING` opcije omogućava da povežete definiciju pogleda sa objektima koji se referenciraju. Prema definiciji, svi objekti koje referencira pogled mogu biti uklonjeni. U ovoj situaciji, pogled će generisati grešku prilikom sledećeg korišćenja. Ukoliko se koristi `WITH SCHEMABINDING` opcija, greška će se javiti kada se pokuša uklanjanje objekta koji referencira pogled. Kada se podaci modifikuju putem definicije pogleda, modifikacije se ne proveravaju u odnosu na bilo koji filter koji se nalazi u `WHERE` klauzuli upita. To omogućava da se izvršavaju modifikacije zapisa u pogledu tako da one ne budu vidljive prilikom narednog korišćenja pogleda.

## Korisnički definisane funkcije

Korisnički definisane rutine se kreiraju korišćenjem Transact-SQL ili .NET Common Language Runtime naredbi. U ovom poglavlju ćemo se fokusirati na kreiranje Transact-SQL korisnički definisanih funkcija. Više informacija o kreiranju funkcija korišćenjem .Net jezika možete pronaći u Poglavlju 18.

Korisnički definisane funkcije mogu da sadrže jedan ili više parametara, izvršavaju izračunavanja ili neke druge operacije, odnosno vraćaju skalarnu vrednost ili rezultujući skup. Funkcije koje vraćaju rezultujući skup smatraju se funkcijama koje vraćaju tabelarne vrednosti i one se koriste u `FROM` klauzuli upita. Skalarne funkcije se mogu koristiti u upitima, `CHECK` ograničenjima, kolonama čije se vrednosti izračunavaju, upravljačkim naredbama, kao i na mnogim drugim mestima.

## Funkcije i snimljene procedure

Korisnički definisane funkcije imaju dosta sličnosti sa snimljenim procedurama, ali postoji i niz razlika. Snimljene procedure su raspoložive u SQL Server okruženju mnogo duže od korisnički definisanih funkcija, i koristi ih veliki broj programera, iako korisnički definisane funkcije predstavljaju mnogo fleksibilniju opciju. Korisnički definisane funkcije imaju brojne prednosti u odnosu na snimljene procedure:

**Modularno programiranje** Korisnički definisane funkcije promovišu višestruko korišćenje Transact-SQL logike u aplikaciji. Korisnički definisane funkcije se mogu referencirati na mnogo više lokacija u odnosu na snimljene procedure.

**Redukovanje mrežnog saobraćaja** Korisnički definisanim funkcijama se može značajno redukovati mrežni saobraćaj, kao što se to čini korišćenjem snimljenih procedura. Funkcija može da sadrži veliki broj naredbi i može da se koristi u nekoliko delova upita.

**Keširanje plana izvršenja** Planovi izvršenja, kreirani na osnovu korisnički definisanih funkcija, keširaju se na isti način kao i kada se radi o snimljenim procedurama. To znači da optimizator upita ne mora ponovo da kreira planove za izvršavanje upita za isti deo koda koji se koristi u više različitih upita.

Dve glavne razlike između snimljenih procedura i korisnički definisanih funkcija su način na koji se one izvršavaju i način na koji vraćaju rezultate nakon izvršavanja. Korisnički definisane funkcije obično obezbeđuju mnogo fleksibilnije izračunavanje u odnosu na snimljene procedure, koje vraćaju skalarne vrednosti ili jedan rezultujući skup podataka. Takođe, primenom korisnički definisanih funkcija, a ne snimljenih procedura, omogućavate da kod bude mnogo pregledniji, razumljiviji i da može mnogo jednostavnije da se održava kada za to postoji potreba.

Snimljene procedure koje vraćaju jedan rezultujući skup podataka obično mogu da se napišu u obliku funkcija. Prilikom izvršavanja snimljene procedure, rezultujući skup se može jednostavno spajati sa drugim tabelama u upitu, a da se pri tome ne zahteva kreiranje privremene tabele, kao što je prikazano u sledećem primeru:

```
CREATE TABLE #temp (…)
INSERT INTO #temp
EXEC uspExample 1, 2;
SELECT * FROM #temp JOIN anotherTable…
```

Korisnički definisana funkcija koja vraća više vrednosti, koje su identične vrednostima koje vraća snimljena procedura, može se referencirati u `SELECT` naredbi, kao što je prikazano u sledećem primeru:

```
SELECT * FROM udfExample(1, 2) JOIN anotherTable…
```

Snimljene procedure koje vraćaju skalarne vrednosti zahtevaju da definišete promenljivu ukoliko želite da se ove vrednosti ponovo koriste u narednim pozivima snimljenih procedura.

U sledećem primeru prikazan je proces neophodan za izvršavanje snimljene procedure, koja prikazuje jedan rezultat, a štampa konačan rezultat:

```
DECLARE @proc1out int,
  @proc2out int;
EXEC proc1 @param1, @param2, @proc1out OUTPUT;
EXEC proc2 @proc1out, @proc2out OUTPUT;
PRINT @proc2out;
```

Funkcije mogu da koriste rezultat neke druge funkcije kao svoj ulazni parametar. Sledeći primer je ekvivalentan korišćenju dve funkcije za izračunavanje:

```
PRINT Function1(Function2(@param1, @param2));
```

## Ponovno podsećanje na ugrađene funkcije

SQL Server obezbeđuje veliki broj ugrađenih sistemskih funkcija. Ove funkcije se koriste prilikom izvršavanja različitih kategorija zadataka, kao što su rad sa stringovima, matematička izračunavanja i korišćenje metapodataka. Korisnički definisane funkcije su veoma slične ugrađenim funkcijama, bar kada se radi o načinu njihovog korišćenja. U brojnim situacijama, korisnički definisane funkcije uključuju neke od sistemskih funkcija u svom telu.

Ugrađene funkcije se mogu kategorizovati u deterministčke i nedeterminističke funkcije. Deterministička funkcija će uvek vratiti isti rezultat za isti skup ulaznih vrednosti i uslova definisanih za bazu podataka. Primer determinističke funkcije može biti `SUBSTRING`; ukoliko navedete iste ulazne parametre, uvek će generisati identičan rezultat:

```
PRINT SUBSTRING('ABCDE', 2, 1);
```

Izvršavanjem prethodnog koda uvek ćete dobiti slovo `B`. `GETDATE()` je primer jedne nedeterminističke funkcije. Rezultat izvršavanja sledećeg primera zavisi od trenutnog datuma i vremena:

```
PRINT GETDATE() + 1;
```

## Skalarne funkcije

Skalarne funkcije izvršavaju određene operacije nad jednim parametrom ili većim brojem parametara, u cilju generisanja jedne vrednosti. Prilikom definisanja funkcija, parametri moraju da se navode između zagrada. Skalarna funkcija može da vraća bilo koji tip podataka, osim tipova text, ntext, cursor i timestamp. Ukoliko funkcija sadrži više od jedne naredbe, morate da koristite `BEGIN` i `END`.

Skalarne funkcije se mogu koristiti u brojnim situacijama, i to u:

- `SELECT` naredbama
  - Listi kolona
  - `WHERE` klauzuli
  - `ORDER BY` listi
  - `GROUP BY` listi
- `SET` klauzuli `UPDATE` naredbi
- `INSERT` naredbama
- `CHECK` ograničenjima
- `DEFAULT` ograničenjima
- Kolonama čije se vrednosti izračunavaju
- Upravljačkim naredbama
- Funkcijama i snimljenim procedurama

U sledećem primeru prikazano je korišćenje skalarne korisnički definisane funkcije, koja vraća kvartal i godinu za posmatrani datum:

```
USE AdventureWorks2008;
GO
```

```
CREATE FUNCTION dbo.Qtr
 (@InDate datetime)
 RETURNS char(9)
AS
BEGIN
  RETURN 'FY' + CAST(YEAR(@InDate) As varchar) +
  '-Q' + CAST(DATEPART(qq, @InDate) AS varchar);
END
GO
```

Ova funkcija formatira povratnu vrednost na način koji je standardan prilikom kreiranja izveštaja. Datumi se konvertuju tako da datum koji pripada prvom kvartalu fiskalne 2008. godine treba da se prikaže kao FY2008-Q1. U sledećem primeru prikazano je korišćenje `PRINT` naredbe:

```
PRINT dbo.Qtr('3/20/2008');
```

Skalarna funkcija se može primenjivati svuda gde je dozvoljeno korišćenje skalarnih izraza. Na primer, u sledećem upitu se koristi funkcija za prikazivanje podataka koji su raspoređeni po kvartalima:

```
SELECT dbo.Qtr(OrderDate) As OrderQuarter,
  SUM(TotalDue) As TotalSales
 FROM Sales.SalesOrderHeader
 GROUP BY dbo.Qtr(OrderDate)
 ORDER BY dbo.Qtr(OrderDate);
```

Skalarnim korisnički definisanim funkcijama možete uraditi mnogo više od pukog formatiranja datuma. Bilo koji skup Transact-SQL naredbi, koji se ne koristi za modifikovanje podataka, može se primenjivati za određivanje skalarne vrednosti koja predstavlja rezultat izvršavanja funkcije. Sledeća funkcija je primer iz AdventureWorks2008 baze podataka. Ona je projektovana da bi se odredile ukupne zalihe u inventaru za posmatrani proizvod.

```
USE AdventureWorks2008;
GO
CREATE FUNCTION dbo.ufnGetTotalInventoryStock
 (@ProductID int)
RETURNS int
AS
BEGIN
 DECLARE @ret int;
 SELECT @ret = SUM(p.Quantity)
 FROM Production.ProductInventory p
 WHERE p.ProductID = @ProductID;
 IF (@ret IS NULL)
  SET @ret = 0;
 RETURN @ret;
END;
```

Pogledajte sledeći primer

```
SELECT ProductNumber, Name,
  dbo.ufnGetTotalInventoryStock(ProductID) As InventoryCount
FROM Production.Product
WHERE ProductNumber LIKE 'EC%'
```

Prethodna funkcija se izvršava za svaku pojedinačnu vrstu koju vraća upit. Ova funkcija može biti veoma efikasna u kombinaciji sa naredbama za upravljanje tokom izvršavanja koda u proceduri, kao što je prikazano u sledećem primeru:

```
IF dbo.ufnGetTotalInventoryStock(@ProductID) < 500
BEGIN
— Izvršavanje operacije povezane za proizvodom za koji ne postoje dovoljne zalihe
END;
```

## Funkcije koje vraćaju vrednosti u tabeli

Funkcije koje vraćaju vrednosti u tabeli su veoma moćno sredstvo za generisanje rezultujućeg skupa podataka. One se mogu koristiti svuda gde je u upitu dozvoljeno korišćenje tabele ili pogleda. Kada se radi o korišćenju, ove funkcije su mnogo fleksibilnije u odnosu na snimljenu proceduru koja vraća rezultujući skup podataka, pošto se rezultujući skupovi funkcija mogu spajati sa drugim tabelama u upitu.

SQL Server definiše dva tipa funkcija koje vraćaju više vrednosti. Linijske funkcije koje vraćaju vrednosti u tabeli su konceptualno slične pogledu sa parametrima. Složene funkcije koje vraćaju vrednosti u tabeli omogućavaju vam da koristite više naredbi za kreiranje rezultujućeg skupa koji se predstavlja u obliku podataka tabele.

#### Linijske funkcije koje vraćaju vrednosti u tabeli

Linijske funkcije koje vraćaju vrednosti u tabeli veoma je jednostavno kreirati. Sadržaj linijske funkcije koja vraća vrednosti u tabeli je `SELECT` naredba sa odgovarajućim parametrima. Povratni tip podataka je uvek tabela, ali je struktura povratne tabele definisana strukturom `SELECT` naredbe. U sledećem primeru prikazana je jedna linijska funkcija koja se koristi za određivanje ukupne prodaje proizvoda za dati `CustomerID` identifikator:

```
USE AdventureWorks2008;
GO
CREATE FUNCTION Sales.ufnSalesByCustomer (@CustomerID int)
RETURNS TABLE
AS
RETURN
(
 SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS Total
 FROM Production.Product AS P
  JOIN Sales.SalesOrderDetail AS SD
  ON SD.ProductID = P.ProductID
  JOIN Sales.SalesOrderHeader AS SH
  ON SH.SalesOrderID = SD.SalesOrderID
 WHERE SH.CustomerID = @CustomerID
 GROUP BY P.ProductID, P.Name
);
GO
```

Obratite pažnju na to da se telo ove funkcije sastoji od jedne `RETURN` naredbe. U sledećem primeru koristi se upit u kome se primenjuje prethodno definisana funkcija:

```
SELECT * FROM Sales.ufnSalesByCustomer(30052);
```

Linijske funkcije ovog tipa predstavljaju veoma moćan alat, koji možete da koristite u situacijama u kojima je neophodno parametrizovanje upita. One omogućavaju mnogo veću fleksibilnost prilikom korišćenja rezultujućeg skupa podataka.

#### SLOŽENE FUNKCIJE KOJE VRAĆAJU VREDNOSTI U TABELI

Složene funkcije koje vraćaju vrednosti u tabeli omogućavaju vam da koristite više Transact-SQL naredbi prilikom kreiranja sadržaja tabele. Ove funkcije predstavljaju veoma moćnu alternativu snimljenim procedurama koje kreiraju rezultujuće skupove izvršavanjem više operacija.

Ovaj tip funkcija omogućava programeru da dinamički ispunjava tabelu u više koraka, koji su identični koracima koji se definišu u snimljenim procedurama, ali se mogu referencirati kao tabele u `SELECT` naredbi.

Prilikom korišćenja složenih funkcija koje vraćaju vrednosti u tabeli, struktura tabele mora da bude definisana u zaglavlju same funkcije. Za tabelu se definiše odgovarajuća promenljiva, a sve operacije modifikovanja podataka se mogu primenjivati samo nad tako definisanom promenljivom.

U sledećem primeru prikazana je funkcija koja je slična `ufnSalesByCustomer` funkciji, koju smo definisali u prethodnom odeljku. Prvo se definiše promenljiva vezana za tabelu, a zatim se ažurira ta promenljiva, kako bi se uključio celokupan inventar proizvoda korišćenjem prethodno kreirane skalarne funkcije. Naredbe neophodne za kreiranje funkcije prikazane su u sledećem kodu:

```
USE AdventureWorks2008;
GO
CREATE FUNCTION Sales.ufnSalesByCustomerMS (@CustomerID int)
RETURNS @table TABLE
( ProductID int PRIMARY KEY NOT NULL,
 ProductName nvarchar(50) NOT NULL,
 TotalSales numeric(38,6) NOT NULL,
 TotalInventory int NOT NULL )
AS
BEGIN
 INSERT INTO @table
 SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS Total, 0
 FROM Production.Product AS P
 JOIN Sales.SalesOrderDetail SD ON SD.ProductID = P.ProductID
 JOIN Sales.SalesOrderHeader SH ON SH.SalesOrderID = SD.SalesOrderID
 WHERE SH.CustomerID = @CustomerID
 GROUP BY P.ProductID, P.Name;

 UPDATE @table
 SET TotalInventory = dbo.ufnGetTotalInventoryStock(ProductID);

 RETURN;
END;
```

Izvršavanje ove funkcije je identično izvršavanju prethodno definisane linijske funkcije:

```
SELECT * FROM Sales. ufnSalesByCustomerMS (30052);
```

# Korišćenje sinonima

*Sinonim* vam omogućava da kreirate alternativne nazive za objekte unutar baze podataka. Ukoliko neki objekat preimenujete, ili se promeni šema objekta, sinonim uvek može da omogući postojećim aplikacijama da nastave sa korišćenjem prethodnih naziva.

Sinonimi mogu da referenciraju objekte u različitim bazama podataka, pa čak i na različitim serverima, korišćenjem naziva objekata koji se sastoje od tri ili četiri dela. Sinonim mora da referencira objekat baze podataka, a ne neki drugi sinonim. Više naziva može da se kreira za jedan objekat baze podataka, sve dok oni direktno ukazuju na odgovarajući objekat baze podataka.

Dok sinonimi oezbeđuju alternativne nazive, dozvole se i dalje održavaju na nivou objekata baze podataka. Korisniku mora da bude omogućeno pristupanje sinonimu, ali sve dok se zahteva pristup odgovarajućem raspoloživom objektu, naredbe koje koriste taj sinonim neće moći da se koriste.

## Zbog čega je potrebno koristiti sinonime?

Sinonime možete koristiti u velikom broju situacija, i to prilikom:

- Preimenovanja objekata.
- Pomeranja objekata u drugu šemu.
- Pomeranja objekata u posebnu bazu podataka.
- Pomeranja objekata na drugi server.
- Potrebe za korišćenjem alternativnih naziva za objekte baze podataka.

Sinonimi mogu referencirati brojne tipove objekata baze podataka i omogućavaju veću fleksibilnost kada se radi o nazivima i lokacijama. Korišćenjem sinonima pojednostavljeno je održavanje baza podataka, a oni omogućavaju kreiranje posebnog sloja apstrakcije u aplikacijama, kojim je omogućeno direktno pristupanje željenim objektima.

## Kreiranje sinonima

Kreiranje sinonima se može realizovati pomoću Transact-SQL naredbi, kao i interfejsa SQL Server Management Studio alata. Kreiranje sinonima korišćenjem Transact-SQL naredbi obavlja se na sledeći način:

```
CREATE SYNONYM shema.nazivSinonima FOR osnovniObjekat;
```

U sledećem primeru prikazano je kreiranje alijasa u Person shemi za HumanResources.Employee tabelu, koja se nalazi u AdventureWorks2008 bazi podataka:

```
USE AdventureWorks2008;
GO
CREATE SYNONYM Person.Employee FOR HumanResources.Employee;
```

Sinonimi se mogu kreirati i za poglede, funkcije i snimljene procedure, što je prikazano u sledećem primeru:

```
CREATE SYNONYM Person.vEmployee FOR HumanResources.vEmployee;
CREATE SYNONYM dbo.SalesByCust FOR Sales.ufnSalesByCustomer;
CREATE SYNONYM Production.getBOM FOR dbo.uspGetBillOfMaterials;
```

Sinonimi se mogu kreirati za objekte u drugim bazama podataka ili na nekim drugim povezanim serverima. U sledećem primeru pretpostavlja se da je definisano povezivanje vlasništva i da je povezani server, pod nazivom server2, podešen na odgovarajući način:

```
USE TempDB;
CREATE SYNONYM dbo.ExampleTbl
FOR AdventureWorks2008.HumanResources.Employees;

USE AdventureWorks2008;
CREATE SYNONYM dbo.RemoteProducts
FOR Server2.AdventureWorksRemote.Production.Product;
```

# Vežbe

**Kreiranje snimljenih procedura.** Snimljene procedure predstavljaju veoma moćan mehanizam za enkapsulaciju procesa vezanih za bazu podataka. Razumevanje kreiranja i izvršavanja snimljenih procedura je veoma značajno za svakoga ko želi da postane dobar administrator baza podataka.

**Vežba** Može li snimljena procedura da referencira objekte koji ne postoje? Može li da referencira kolone koje ne postoje u tabelama?

**Kreiranje pogleda.** Pogledi su upiti koji su smešteni kao objekti i koji se referenciraju kao tabele. Kreiranje pogleda vam omogućava dodatnu fleksibilnost prilikom kreiranja i skladištenja upita koji se definišu nad bazomm podataka.

**Vežba** Koje su tri glavne prednosti korišćenja pogleda?

**Kreiranje korisnički definisanih funkcija.** Samostalno kreiranje funkcija vam omogućava veću fleksibilnost, bilo da se obrađuju skalarne vrednosti ili rezultujući skupovi podataka.

**Vežba** Koje su glavne prednosti korišćenja funkcija koje vraćaju rezultate u obliku tabela u odnosu na snimljene procedure koje vraćaju rezultujuće skupove podataka?