VERİ TABANI VE YÖNETİM SİSTEMLERİ

Dr. Öğretim Üyesi

Alper Talha KARADENİZ

GENEL DERS İÇERİĞİ

1.Fonksiyonlar

2.Hata Yönetimi

3.Hata Mesajları

4. TRY-CATCH

5. Raiserror

6. Throw

7. Cursor

FONKSİYONLAR

Veritabanı sistemlerinde fonksiyonlar (functions), belirli bir işlemi gerçekleştiren ve bir değer döndüren SQL nesneleridir. SQL Server gibi sistemlerde fonksiyonlar, genellikle hesaplama, veri dönüştürme, filtreleme gibi işlemler için kullanılır. Fonksiyonlar, kod tekrarını azaltır ve daha okunabilir SQL sorguları yazmamıza yardımcı olur.

FONKSİYONLARIN ÖZELLİKLERİ

- Her zaman bir değer döndürür (tek bir sonuç).
- Parametre alabilirler.
- SELECT içinde kullanılabilirler.
- Fonksiyonlar veritabanı içinde tanımlanır, tekrar tekrar çağrılabilir.
- Yan etki yaratmazlar (genellikle veri ekleme, silme işlem yapmazlar)

FONKSİYON NE ZAMAN KULLANILIR?

- Aynı işlemi tekrar tekrar yapıyorsan (örneğin KDV, yaş hesaplama vs.)
- SELECT içinde özel bir hesaplama yapman gerekiyorsa
- Kodun daha temiz ve anlaşılır olmasını istiyorsan
- Veritabanı içinde modüler yapı istiyorsan

FONKSİYONLARI DÜZENLEMEK

Diğer nesnelerde olduğu gibi ALTER komutu ile düzenlenebilir

```
ALTER FUNCTION ort_maas()
RETURNS INT
AS
BEGIN
DECLARE @maas INT
select @maas=SUM(maas) from personel
RETURN @maas
END

SELECT dbo.ort_maas()
```

FONKSİYONLARI SİLMEK

Fonksiyon silmek için DROP komutu kullanılır.

```
DROP FUNCTION dbo.ort_maas
```

ÖRNEKLER

KDV HESAPLAMA

```
CREATE FUNCTION fn_kdv_hesapla (@fiyat DECIMAL(10,2))
RETURNS DECIMAL(10,2)
AS
BEGIN
 RETURN @fiyat * 1.20; -- %20 KDV eklendi
END;
```

KULLANIM: SELECT dbo.fn_kdv_hesapla(100); -- *Sonuç: 120.00*

Verilen bir fiyata %20 KDV ekleyerek geri döner.

ÖRNEKLER

YAŞ HESAPLAMA

```
CREATE FUNCTION fn_yas_hesapla (@dogum_tarihi DATE)
RETURNS INT
AS
BEGIN
 RETURN DATEDIFF(YEAR, @dogum_tarihi, GETDATE());
END;
```

KULLANIM: SELECT dbo.fn_yas_hesapla('2000-05-10'); -- *Sonuç: 25 (tarihe göre değişebilir)*

Verilen doğum tarihine göre bugünkü yaş hesaplanır.

ÖRNEKLER

AKTİF PERSONELLERİ LİSTELE

```
CREATE FUNCTION fn_aktif_personeller()
RETURNS TABLE
AS
RETURN (
 SELECT * FROM personel WHERE durum = 'aktif'
);
```

KULLANIM: SELECT * FROM dbo.fn_aktif_personeller();

Aktif olan personelleri tablo şeklinde döner.

ÖRNEKLER

MAAŞA GÖRE PERSONEL

```
CREATE FUNCTION fn_maasa_gore_personel (@minMaas INT)
RETURNS @sonuc TABLE (
 ad NVARCHAR(50),
 maas INT) AS BEGIN
 INSERT INTO @sonuc
 SELECT ad, maas FROM personel WHERE maas >= @minMaas;
 RETURN;
END;
```

KULLANIM: SELECT * FROM dbo.fn_maasa_gore_personel(15000);

Verilen maaş değerinden yüksek maaş alan personelleri döner.

ÖRNEKLER

AD SOYAD BİRLEŞTİRME

```
CREATE FUNCTION fn_ad_soyad (@ad NVARCHAR(50), @soyad
NVARCHAR(50))
RETURNS NVARCHAR(100)
AS
BEGIN
 RETURN @ad + ' ' + @soyad;
END; KULLANIM: SELECT dbo.fn_ad_soyad('Alper', 'Karadeniz');
```

-- Sonuç: Alper Karadeniz

Ad ve soyad değerlerini birleştirerek tek bir metin döner.

ÖRNEKLER

CİNSİYETE GÖRE PRİM DEĞERİ

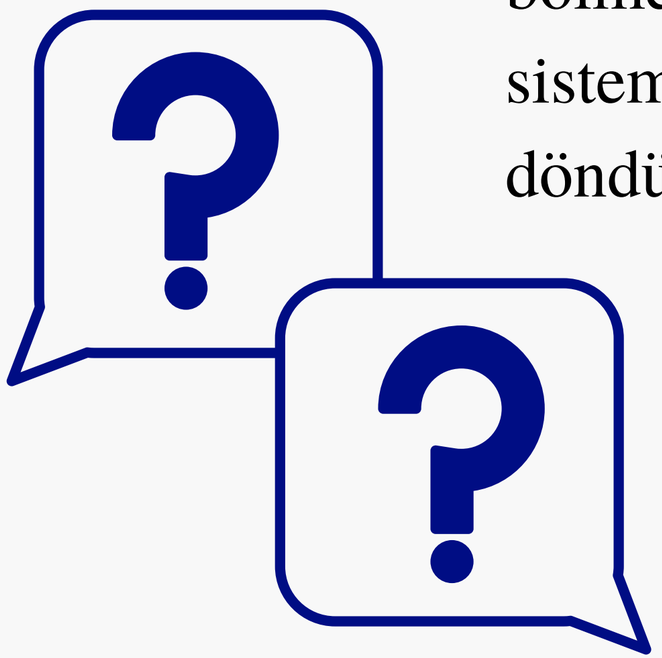
```
CREATE FUNCTION top_prim(@cins varchar(50)=NULL)
RETURNS INT
AS
BEGIN
DECLARE @prim INT select @prim =SUM(prim) from personel
WHERE cinsiyet=@cins
RETURN @prim
END
```

SELECT dbo.ort_maas('K')

Dışarıdan girilen cinsiyete göre toplam prim değerini döndürür

Hata Yönetimi Nedir? (Error Handling)

Hata yönetimi, bir SQL komutu veya işlem sırasında oluşabilecek hataları yakalama, kontrol etme ve yönetme sürecidir. Özellikle veritabanı işlemlerinde beklenmeyen durumlar (örneğin sıfıra bölme, veri tipi uyuşmazlığı, tablo bulunamaması vb.) oluştuğunda sistemin çökmesini önlemek ve kullanıcıya anlamlı mesajlar döndürmek için hata yönetimi yapılır.



Hata Yönetimi Neden Gerekli?

- Uygulamanın beklenmedik şekilde kesilmesini engellemek
- Kullanıcıya daha açıklayıcı hata mesajları göstermek
- Sistem güvenliğini artırmak
- Veri bütünlüğünü korumak (örneğin TRY-CATCH + TRANSACTION ile)

Hata Mesajları

Message_ID

- 0 - 49.999: Sistem hata mesajı kodları için ayrılmıştır.
- 50.000: RAISERROR fonksiyonu ile üretilen anlık hata mesajları için ayrılmıştır
- .50.001 - ... : Kullanıcı tanımlı mesajlar için ayrılmıştır.

Language_ID

Sistemdeki dil grup kodu. Hatanın hangi dilde olduğunu gösterir. (1033 = İngilizce)

Hata Mesajları

Severity

1 - 10: Kullanıcıdan kaynaklanan bilgi içerikli hatalardır.

11 - 16: Kullanıcının TRY/CATCH bloğu ile yönetebileceği hatalardır.

17: Disk ya da diğer kaynakların tükendiği durumlarda oluşur. TempDB'nin dolu olması gibi. TRY/CATCH blokları ile yakalanabilir.

18: Kritik, dahili ve sistem yöneticisini ilgilendiren hatadır.

19: WITH LOG özelliğinin kullanılması gerekir. Hata NT ya da Windows Event Log'da gösterilecektir. TRY/CATCH bloğu ile yakalanabilir.

20 - 25: Tehlikeli hatalardır. Kullanıcı bağlantısı sonlandırılır. WITH LOG uygulanması gerekir. Event Log'da görüntülenebilir.

@@ERROR: *OLUŞAN SON HATANIN KODUNU YAKALAMAK*

SQL Server'da, bir işlem (örneğin bir INSERT, UPDATE, DELETE) yapıldığında bu işlemin başarılı olup olmadığını kontrol etmek için @@ERROR adlı sistem değişkeni kullanılır.

- Eğer işlem başarılıysa @@ERROR değeri 0 olur.
- Eğer bir hata oluşursa @@ERROR, o hataya ait hata kodunu tutar.

Ancak önemli bir nokta şudur:

@@ERROR değeri yalnızca son çalıştırılan komutun sonucunu verir.

Yeni bir komut çalıştırıldığında, @@ERROR otomatik olarak güncellenir ve önceki değeri kaybolur.

```sql
DECLARE @sayi INT, @hataKod INT;
SET @sayi = 0
SELECT maas /@sayi
FROM personel
WHERE personel_no = 5
SET @hataKod = @@ERROR
IF @hataKod=8134
BEGIN
PRINT CAST(@hataKod AS VARCHAR) + ' Nolu sıfıra bölünme hatası.' ;
END
ELSE
BEGIN
PRINT CAST(@hataKod AS VARCHAR) + ' Nolu bilinmeyen bir hata oluştu.';
END
```

HATA KONTROLÜ VE TRY-CATCH

TRY...CATCH, SQL Server'da hataları kontrol altına almak ve sistemin çökmesini önlemek için kullanılan bir yapıdır.

- TRY bloğu, hataya neden olabilecek işlemleri içerir. Eğer burada bir hata oluşursa, işlem durdurulur ve doğrudan CATCH bloğuna geçilir.
- CATCH bloğu ise hatayla nasıl başa çıkılacağını belirler: örneğin kullanıcıya mesaj göstermek, işlemi geri almak (ROLLBACK), loglamak vs.

```
BEGIN TRY
 { sql bloğu } // yapılacak işlem burada yazılır
END TRY
BEGIN CATCH
 { sql ifadeleri } // hata mesajı burada yazılır
END CATCH;
```

```
BEGIN TRY
DECLARE @sayi INT=8/0
END TRY

BEGIN CATCH
SELECT ERROR_LINE(),
ERROR_MESSAGE(),
ERROR_NUMBER(),
ERROR_SEVERITY();
END CATCH
```

```sql
DECLARE @sayi1 INT = 5
DECLARE @sayi2 INT = 0
DECLARE @sonuc INT
BEGIN TRY
SET @sonuc = @sayi1 / @sayi2
END TRY
BEGIN CATCH
PRINT CAST(@@ERROR AS VARCHAR) + ' no lu hata oluştu'
END CATCH;
```

EKRAN ÇIKTISI: «8134 no lu hata oluştu»

```sql
SELECT * FROM sys.messages WHERE message_id = 8134 AND language_id = 1055;
```

Hata Fırlatma (Exception Throwing) Nedir?

SQL Server'da bir hata oluştuğunda, bu hatayı sadece yakalamak (TRY-CATCH ile) yeterli olmayabilir. Bazen hatayı manuel olarak da fırlatmamız gerekir. Yani hata oluşmamış olsa bile biz sistemin "bir hata varmış gibi davranmasını" isteyebiliriz. İşte bu gibi durumlarda RAISERROR ve THROW komutları kullanılır.

RAISERROR NEDİR?

RAISERROR, SQL Server’ın eski sürümlerinden beri kullanılan, hata mesajı üretmek ve tetiklemek için kullanılan bir ifadedir.

```
RAISERROR('Geçersiz işlem yapıldı!', 16, 1);
```

RAISERROR iki farklı amaç ile kullanılabilir. Bunlar;

- Sistemde var olan hata mesajları ile bir hata meydana getirmek.
- Anlık oluşturulan bir hata mesajı ile bir hata meydana getirmek.

THROW NEDİR?

THROW, SQL Server 2012 ile gelen modern hata fırlatma yöntemidir. RAISERROR'a göre kullanımı daha basittir ve TRY-CATCH yapısıyla daha iyi çalışır.

THROW 51000, 'İzin verilmeyen işlem!', 1;

ya da TRY-CATCH içinde basitçe:

THROW; -- *son hatayı yeniden fırlatır*

CURSOR

Cursor’ler, veri kümesini ele alarak her seferinde bir kayıt üzerinde işlem yapabilmeyi sağlayan yöntemdir. Cursor’ler veritabanında sürekli kullanmanız gereken bir özellik olmasa da, gerektiği durumlarda birçok faydalı işleve sahiptir. Az da olsa kullanmanız gereken faydalı bir özelliktir. Cursor, varsayılan olarak sadece ileri doğru işlem yapar. Geriye doğru işlem yapabiliyor olsa da, ana kullanım yöntemi ileriye doğrudur. Ayrıca sadece ileri doğru çalışan Cursor ile, kaydırma yapabilen Cursor’lar arasında önemli performans ve hız farkı vardır.

Cursor’ların en yaygın kullanım amaçları:

Kayıtlar Arasında Gezinmek

- Bir sorgudan dönen kayıt kümesi (result set) üzerinde tek tek dolaşmak,
- İlk kayda, son kayda, önceki ya da sonraki kayda geçmek gibi işlemleri gerçekleştirmek için kullanılır.

Sorgu Sonucundaki Satırlarda Güncelleme Yapmak

- Belirli koşullara uyan satırlarda manuel olarak güncelleme, silme veya özel işlem yapılmak isteniyorsa cursor kullanılır.

Stored Procedure veya Trigger İçinde Satır Satır İşlem Yapmak

- Trigger veya stored procedure içinde çalıştırıldığında cursor’lar sayesinde her satıra ayrı müdahale edilebilir.
- Özellikle toplu güncellemelerde ya da karmaşık kontrollerde tercih edilir.

ÖRNEK

Cursor ile satır okurken her kolonu bir değişkene atarız. Bu yüzden önce değişkenler tanımlanır.

DECLARE @personel_id INT, @maas INT;

Cursor'a bir SELECT sorgusu verilir. Bu sorgu, üzerinde işlem yapacağımız satırları belirler.

```
DECLARE cursor_maas_zam CURSOR FOR
SELECT personel_id, maas
FROM personel;
```

NOT: SELECT sorgusunun getirdiği kolon sayısı ile tanımladığın değişken sayısı bire bir eşleşmeli

Koşullu veri almak istiyorsan burada WHERE kullanabilirsin.

ÖRNEK

Tanımlanan cursor belleğe yüklenir ve kullanıma hazır hâle getirilir.

OPEN cursor_maas_zam;

İlk veriyi almak için FETCH NEXT kullanılır ve değerler değişkenlere atanır.

FETCH NEXT FROM cursor_maas_zam INTO @personel_id, @maas;

ÖRNEK

Satır sonuna kadar FETCH işlemini döngüde tekrarlarsın. @@FETCH_STATUS = 0 olduğu sürece satır var demektir.

```
WHILE @@FETCH_STATUS = 0
BEGIN
UPDATE personel
SET maas = @maas + 1000
WHERE personel_id = @personel_id;

   Sonraki satıra geç
 FETCH NEXT FROM cursor_maas_zam INTO @personel_id, @maas;
END;
```

İşin bitince cursor’ı kapatman gerekir, aksi takdirde bellek boşuna işgal edilir.

CLOSE cursor_maas_zam;

Bellekteki tüm tanımlar kaldırılır. Artık bu cursor kullanılamaz.

DEALLOCATE cursor_maas_zam;

Cursorda Komutlarla Adım Adım Gezinmek

❖Bir kayıt ileri hareket ettirir.

FETCH NEXT FROM cursor_maas_zam;

❖Önceki kayda gider.

FETCH PRIOR FROM cursor_maas_zam;

❖İlk kayda gider.

FETCH FIRST FROM cursor_maas_zam;

❖Son kayda gider.

FETCH LAST FROMcursor_maas_zam;

❖Bulunulan yerden 2 kayıt ileri konumlanır.

FETCH RELATIVE 2 FROM cursor_maas_zam

❖Baştan 2. kayda konumlanır.

FETCH ABSOLUTE 2 FROM cursor_maas_zam;