# STORED PROCEDURE'LER

# 11

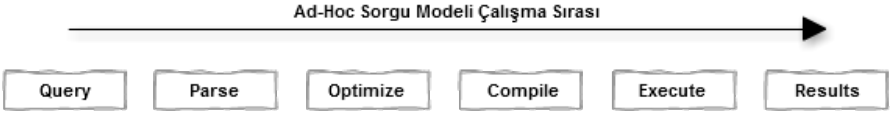
Kod bloklarının, bir kez yazılarak derlendikten sonra SQL Server'da prosedür hafızasında tutulan, tekrar çalıştırıldıklarında ise, ilk sorgu sırasında oluşturulan sorgu ve çalıştırma planı hazır olduğu ve hafızada tutulduğu için daha hızlı sonuç döndüren SQL Server nesneleridir. Aynı zamanda bir çok farklı ara işlemi tekrarlamadığı için, ağ trafiği ve sistem kaynaklarında performans artışı sağlar.

Stored Procedure'ler veritabanı programlamanın en önemli konularından biridir. Veritabanı geliştiricisine SQL'in kısıtlı imkanlarından kurtularak T-SQL ve SQL Server'ın mimari gücünü tam olarak kullanabilme imkanı verir. Stored Procedure'ler kısaca SPROC, SP ya da PR olarak isimlendirilirler.

Programlara yetenekler kazandırma konusunda yardımcı olan sproc'lar, SQL Server'da parametreli ya da parametresiz olarak geliştirilebilir. Dışarıdan bir parametre alabileceği gibi, aldığı parametreyi dışarı da iletebilir.

Normal sorgular (Ad-Hoc) ile Stored Procedure'lerin çalışma planlarını inceleyelim.

**Normal sorgular;**



Normal sorgularda 6 işlem aşaması vardır. Bunlar;

- **Query** (*Sorgu*)

  Oluşturulacak SQL sorgusunun hazırlanma aşamasıdır.

- **Parse** (*Ayrıştırma*)

  Veritabanı motoruna iletilen SQL sorgularının SQL ve T-SQL kurallarına uygunluğunu kontrol eder. Sorgunun çalışması ya da çalışmaması konusu bu kısımda incelenmez ve önemsenmez. Veritabanında gerçekte var olmayan bir tablo için hazırladığınız `SELECT` sorgusu, bu aşamada hata vermez. Sadece sorgunun SQL kurallarına uygunluğu, hatalı kod yazımı yapılıp yapılmadığı gibi söz dizimi kurallarına bakılır.

- **Optimize** (*İyileştirme*)

  Optimize için, SQL Server'ın maliyet hesaplama kısmı denebilir. SQL Server, bir sorguyu çalıştırmadan önce, en doğru sonuca, en hızlı ve performanslı şekilde nasıl ulaşabileceğini hesaplar. Bunun için onlarca farklı sorgu yapısı oluşturarak bunların her birinin performansını derecelendirebilir. Sonuç olarak ise en iyi optimize edilmiş sorgu planını hazırlar.

- **Compile** (*Derleme*)

  Optimize kısmında hazırlanan sorgu planına göre hazırlanan kodların derlenme aşamasıdır. Bu işlemden sonra artık kodlar çalıştırılmaya hazırdır.

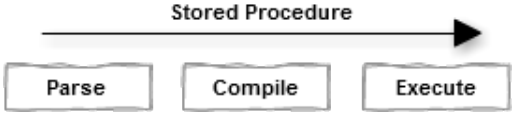
- **Execute** (*Çalıştırma*)

  Tüm süreçlerin sonunda sorgu planı hazır olan kodların çalıştırılacağı kısımdır.

- **Results** (*Sonuçlar*)

  Execute ile kodların çalıştırılması sonrasında, veritabanından alınacak veri varsa, bunların kullanıcıya görüntülendiği kısımdır.

Stored Procedure'ler;



- **Parse** (*Ayrıştırma*)

  Veritabanı motoruna iletilen SQL sorgularının SQL ve T-SQL kurallarına uygunluğunu kontrol eder. Sorgunun çalışması ya da çalışmaması konusu bu kısımda incelenmez ve önemsenmez. Veritabanında gerçekte var olmayan bir tablo için hazırladığınız `SELECT` sorgusu, bu aşamada hata vermez. Sadece sorgunun SQL kurallarına uygunluğu, hatalı kod yazımı yapılıp yapılmadığı gibi söz dizimi kurallarına bakılır.

  **Sorgu Ağacı** (*Query Tree*) ya da **Sıra Ağacı** (*Sequence Tree*) denilen yapı ortaya çıkarılır.

  Parse işlemi sırasında prosedürün, ismi `sysobjects` tablosuna, kaynak kodları `syscomments` tablosuna kaydedilir.

- **Compile** (*Derleme*)

  Parse işleminde oluşturulan sorgu ağacı kullanılarak çalıştırma planı (execution plan) çıkartılır. Çalıştırma planında sorgu için gerekli tüm hak, denetim ve ilgili nesne denetimleri yapılır.

- **Execute** (*Çalıştırma*)

  Compile işleminde oluşturulan çalıştırma planı kullanılarak yapılması istenen işlemler gerçekleştirilir. İstekler, prosedür içerisindeki sorgu yapılarına göre ilgili yöneticilere iletilir. (`SELECT` sorgusu DML yöneticisine iletilir)

## Stored Procedure'lerin Faydaları

Normal sorguların performans olarak Stored Procedure'lere göre düşüktür. Bunun sebebi SQL sorgularının hazırlanması ve çalıştırılana kadar olan süreçlerin sürekli tekrarlanıyor olmasıdır. Ancak bu sadece işin SQL Server içerisindeki durum. Peki, bu sorguların çalıştırıldığı istemci ve ağ araçları açısından durum nedir?

Geliştirilen bir veritabanı istemci uygulamasında onlarca farklı form ve arayüz de yüzlerce, belki de binlerce satır SQL / T-SQL kod bloklarının olması muhtemeldir. Veritabanı yapısının ya da istemcilerdeki SQL kodlarının değiştirilmesi, güncellenmesi gibi bakım işlemlerindeki olası sorunları düşünebiliyor musunuz?

Tüm form ve nesnelerdeki SQL sorgularını sırayla incelemek ve yönetmek zorunluluğu ortaya çıkar. Ayrıca bu tür Ad-Hoc sorgularda, güvenlik büyük bir sorundur. Hem SQL sorgu güvenliği, hem de istemcilerden gelecek SQL Injection gibi saldırı sorgularının filtrelenmesi, kontrol edilmesi gibi sorunlar her zaman bir felakete yol açabilir.

Nesne yönelimli programlama yapan geliştiriciler şüphesiz Stored Procedure'lerin faydalarını daha hızlı kavrayabilirler. Çünkü, Stored Procedure'ler doğru kullanıldığında, fonksiyonel programlama yapılarak uygulamanın geliştirilebilirliği, hız, taşınabilirliği ve anlaşılabilirliğini büyük oranda artıracaktır.

Profesyonel uygulamalarda, tüm SQL sorgularının sadece veritabanı içerisinde yapılması gerekir. İstemci tarafında sadece prosedür ya da view adı gibi isimlendirmeleri kullanarak, tüm işi SQL Server'a yıkmak en doğru yol olacaktır. SQL Server, bir veri işleme ve yönetim platformu olarak güvenlik, performans ve yönetim gibi sorunları çözmüşken, sorguları SQL Server'ın dışında bir yerden yönetmeye çalışmak doğru bir yaklaşım olmayacaktır.

# Stored Procedure Türleri

SQL Server'da Stored Procedure'ler kullanım amaçları ve özelliklerine göre 4'e ayrılır.

## Extended Stored Procedure'ler

Eski bir programlama özelliği olduğu için genel olarak eski uygulamalarda görülebilir. Master veritabanında tutulurlar. C, C++ gibi diller ile geliştirilerek DLL'e derlenirler. COM arayüzünde çalışırlar. Extended Procedure'lerin baş harflerin kısaltması olarak **`xp_`** ya da diğer prosedürlerde kullanıldığı gibi **`sp_`** ön ekini alırlar.

**`xp_cmdshell`**, bu prosedürlere örnek verilebilir. **`xp_cmdshell`** Stored Procedure, SQL Server içerisinden komut satırına erişim sağlar.

Bu XP, çok faydalı olmakla birlikte kullanımında dikkatli olunmalıdır. **`xp_cmdshell`** SQL Server'da varsayılan olarak aktif değildir.

Öncelikle **xp_cmdshell**'i aktif edelim.

```
EXEC sp_configure 'show advanced options',1
GO
RECONFIGURE
GO
EXEC sp_configure 'xp_cmdshell',1
GO
RECONFIGURE
GO
```

Sorguyu çalıştırdıktan sonra, artık **xp_cmdshell** kullanılabilir.

**sp_help** ya da **sp_helptext** kullanılarak **xp_cmdshell** hakkında bilgi alınabilir.

**xp_cmdshell**'i kullanarak **C:** kök dizini içerisindeki dosyaları listeleyelim.

```
master.dbo.xp_cmdshell 'dir';
```

| | output |
|---|---|
| 1 | C sürücüsündeki birimin etiketi yok. |
| 2 | Birim Seri Numarası: 12BB-1AAB |
| 3 | NULL |
| 4 | C:\Windows\system32 dizini |
| 5 | NULL |
| 6 | 11.01.2013 21:10 <DIR> . |
| 7 | 11.01.2013 21:10 <DIR> .. |
| 8 | 21.11.2010 14:35 <DIR> 0409 |
| 9 | 02.01.2013 00:40 <DIR> 1033 |
| 10 | 23.08.2012 12:54 322.560 aaclient.dll |
| 11 | 21.11.2010 05:24 3.745.792 accessibilit... |
| 12 | 14.07.2009 03:24 39.424 ACCTRES.dll |

Bilgisayarımın **C:** dizini içerisinde **sql_files** klasörü ve içerisinde de **.sql** uzantısına sahip dosyalar bulunuyor. Bu klasör içerisinde sadece **.sql** uzantısına sahip olanların tümünü listeleyelim.

```
EXEC master.dbo.xp_cmdshell 'dir C:\sql_files\*.sql';
```

| | output |
|---|---|
| 1 | C sürücüsündeki birimin etiketi yok. |
| 2 | Birim Seri Numarası: 12BB-1AAB |
| 3 | NULL |
| 4 | C:\sql_files dizini |
| 5 | NULL |
| 6 | 09.01.2013 10:30 6 Having.sql |
| 7 | 05.01.2013 13:34 756 IN.sql |
| 8 | 09.01.2013 21:04 496 Insert_Ile_Defa... |
| 9 | 05.01.2013 12:06 1.843 LIKE.sql |
| 10 | 05.01.2013 18:10 739 Null_Islemleri.sql |
| 11 | 10.01.2013 01:03 5.568 SQL_Function... |

**xp_cmdshell** ile işletim sistemi üzerinde birçok kritik işlem yapılabilir.

Bir dosya içerisindeki veriyi bir başka dosyaya kopyalayalım.

**copy1** ve **copy2** adında iki not defteri dokümanı oluşturalım. Daha sonra, **copy1.txt** içerisine bazı veriler girerek aşağıdaki sorguyu çalıştıralım.

```
EXEC master.dbo.xp_cmdshell 'copy D:\copy1.txt D:\copy2.txt';
```

| | output |
|---|---|
| 1 | 1 dosya kopyalandı. |
| 2 | NULL |

Bu işlem sonucunda **copy1.txt** içerisindeki veri **copy2.txt** içerisine kopyalanacaktır.

Microsoft, eski olan bu teknolojiyi SQL Server'ın takip eden versiyonlarında kaldıracaktır. Uygulamalarınızda Extended Strored Procedure'leri kullanmamanız önerilir.

## CLR Stored Procedure'ler

.NET mimarisinin gücünü SQL Server içerisinde kullanmaya olanak tanıyan bu teknolojiyi destekleyen tüm programlama dilleri ile SQL Server ortamında çalışan CLR Stored Procedure'ler oluşturabilirsiniz.

CLR konusu, daha sonra ayrı bir bölüm olarak derinlemesine incelenecektir.

## Sistem Stored Procedure'leri

Master veritabanında tutulan, genellikle **sp_** ön eki alan bu prosedürler ile SQL Server içerisinde birçok özellik ve nesneler hakkında bilgi almak için kullanılır.

Bu sistem prosedürlerine örnek olarak şunlar verilebilir; **sp_help**, **sp_helpdb**, **sp_helptext** ve bunlar gibi daha birçok sistem prosedürü vardır.

## Kullanici Tanimli Stored Procedure'ler

T-SQL programcılarının, yani bizim geliştirdiğimiz stored procedure'lerdir. Bu bölümde işleyeceğimiz konular Kullanıcı Tanımlı SP tipindeki Stored Procedure'leri kapsamaktadır.

# Stored Procedure Oluşturmak

Ürünlerin belirli sütunlarının listesini veren bir **sproc** yazalım.

```
USE AdventureWorks
GO
CREATE PROC pr_UrunleriGetir
AS
SELECT ProductID, Name, ProductNumber, ListPrice FROM Production.Product;
```

Sorgularınızı **AS** ifadesinden sonra **BEGIN...END** bloğu içerisinde de yazabilirsiniz. Bu şekilde kullanım Oracle'da bir zorunluluk olsa da SQL Server'da zorunlu kullanım değildir.

```
BEGIN
     SELECT ProductID, Name,
     ProductNumber, ListPrice
     FROM    Production.Product;
END;
```

Oluşturduğumuz sproc'un kaynak koduna erişmek için;

```
sp_helptext 'pr_UrunleriGetir';
```

| | Text |
|---|---|
| 1 | CREATE PROC pr_UrunleriGetir |
| 2 | AS |
| 3 | SELECT ProductID, Name, ProductNumber, ListPrice... |

# Stored Procedure İçin Gerekli İzin ve Roller

Sproc oluşturabilmek için aşağıdaki izin ya da rollere ihtiyaç vardır.

Roller;

- **sysadmin**
- **db_owner**
- **ddl_admin**

Bu rollere sahip olmadan da sproc oluşturulabilir. Ancak bu durumda kullanıcıya **CREATE PROCEDURE** izninin verilmiş olması gerekir.

## Stored Procedure İçin Kısıtlamalar

Her nesnenin işlevi doğrultusunda bazı kurallara uyması, kısıtlarının olması gerekir. Sproc'lar da belirli kısıtlamalara tabidir.

Bir sproc şu ifadeleri içeremez

- **CREATE PROCEDURE**
- **CREATE DEFAULT**
- **CREATE RULE**
- **CREATE TRIGGER**
- **CREATE VIEW**

Bir sproc yukarıdaki ifadeleri içeremez. Ancak bir başka sproc'tan ya da view, fonksiyon, tablo gibi hemen hemen her nesneden veri alabilir.

## Stored Procedure'ü Çalıştırmak

Bir sproc yazıldıktan sonra sadece parse işlemi gerçekleşir. İlk çalışma anında ise **compile** (*derleme*) ve **execute** (*çalıştırma*) işlemleri gerçekleşir. Bir sproc'da gerçekte olmayan bir tablonun adı geçiyorsa, bununla ilgili hatayı ilk çalıştırma anında alırız.

Bir sproc birden farklı şekilde çalıştırılabilir. Sproc, bulunduğu batch'in ilk ifadesi ise sadece adını yazmak çalıştırmak için yeterlidir. Ancak batch'in ilk ifadesi değilse **EXEC** ya da **EXECUTE** deyimiyle çalıştırılabilir.

3 farklı kullanım;

- **sproc_name;**
- **EXEC sproc_name;**
- **EXECUTE sproc_name;**

## NOCOUNT Oturum Parametresinin Kullanımı

**NOCOUNT**, bir oturum parametresidir. **NOCOUNT**'un kullanımı şu şekildedir:

```
SET NOCOUNT {ON   |   OFF}
```

SQL Server'da her işlemden sonra etkilenen kayıt sayısı hesaplanır ve mesaj olarak geri döndürülür. Bu hesaplama sırasında fazladan kaynak tüketimi gerçekleşir ve bu durum performansı olumsuz yönde etkiler.

Bu özelliğin faydaları olduğu gibi programlama tarafında genel olarak kullanılmasına gerek yoktur. Bu nedenle kaybedilecek performansın geri kazanılması için **NOCOUNT** parametresinin kullanılması önerilir.

**NOCOUNT ON**

```
Command(s) completed successfully.
```

**NOCOUNT OFF**

```
(24 row(s) affected)
```

Sorgularınızın sonucunda kaç kaydın etkilendiği ile ilgilenmiyorsanız, kaç satır etkilendiğini öğrenmenize gerek yoktur. Sorgulardan önce oturum parametresini şu şekilde açarak **NOCOUNT** parametresini kullanabilirsiniz.

```
SET NOCOUNT ON
```

Artık sorgu sonucunda kaç satır etkilendiğini görmeyeceksiniz. Bu özellik SQL Server'da gösterilen mesaj ile ilgilidir. Programatik olarak sorgu sonucunda kaç satır etkilendiğini öğrenmek için **@@ROWCOUNT** global parametresini kullanabilirsiniz. **NOCOUNT** özelliğini aktif etmeniz bu parametreden alacağınız veriyi etkilemez. Yani global parametre ilgili veriyi taşıyarak bize kaç satır etkilendiğini vermeye devam edecektir.

**NOCOUNT** oturum parametresini bir sproc'da şu şekilde kullanarak performansı artırabilirsiniz.

```
CREATE PROC sp_procName
AS
     SET NOCOUNT ON
     -- Stored Procedure'de işlemi gerçekleştirecek sorgu.
GO
```

Bir sproc'da bunu yapabileceğiniz gibi herhangi bir sorgu kullanımında da aynı şekilde kullanabilirsiniz.

```
SET NOCOUNT ON
SELECT * FROM Production.Product;
```

Eğer sorgunuz sonucunda **NOCOUNT** özelliğinin kapatılmasını isterseniz, sorgu bitiminin bir alt satırında şu şekilde **NOCOUNT** özelliğini kapatabilirsiniz.

```
SET NOCOUNT OFF
```

# Stored Procedure'lerde Değişiklik Yapmak

Sproc'ları değiştirmek için **ALTER** deyimi kullanılır.

**Söz Dizimi:**

```
ALTER PROC[EDURE] prosedure_ismi
[WITH seçenekler]
AS
     -- T-SQL sorgusu.
GO
```

Değişiklik yapmak istediğimiz sproc, önceki örneğimizdeki **pr_UrunleriGetir** olsun. Var olan bu sproc'un kaynak kodunu görebilmek için **sp_helptext**'i kullanalım.

```
sp_helptext 'pr_UrunleriGetir';
```

| | Text |
|---|---|
| 1 | CREATE PROC pr_UrunleriGetir |
| 2 | AS |
| 3 | SELECT ProductID, Name, ProductNumber, ListPrice... |

Sorgu ekranına gelen kaynak kodları incelediğimizde **SET NOCOUNT** kullanmadığımızı fark ediyoruz. Şimdi **ALTER** deyimini kullanarak bu sproc'a **NOCOUNT** özelliği kazandıralım.

```
ALTER PROC pr_UrunleriGetir
AS
SET NOCOUNT ON
 SELECT ProductID, Name,
 ProductNumber, ListPrice
 FROM       Production.Product;
SET NOCOUNT OFF
```

# Stored Procedure'leri Yeniden Derlemek

Veritabanı geliştirmeleri sırasında tablo yapıları ve indekslerin değiştirilmesi söz konusu olabilir. Bu gibi durumlarda sproc'ların **çalıştırma planı** (*execution plan*), **tablo** ve **indeks yapısı** farklı iken, derlendiği için performans olarak olumsuz durumlar oluşabilir.

Sproc'ların bu gibi sorunlarla karşılaşmamaları için yeniden derlenmesi gerekir. Bu derleme işlemi ile birlikte yeni tablo ve indeks yapısına göre çalıştırma planı oluşturulacağı için performans olarak faydalı olacaktır.

Bir sproc'un her çalıştırılmada yeniden derlenmesi için şu ifade kullanılır.

```
CREATE PROC prosedure_ismi
WITH RECOMPILE
AS
 -- SQL sorgusu..
GO
```

**pr_UrunleriGetir** isimli sproc'u her çalıştırılmada yeniden derlenecek şekilde değiştirelim.

```
ALTER PROC pr_UrunleriGetir
WITH RECOMPILE
AS
SET NOCOUNT ON
 SELECT ProductID, Name,
 ProductNumber, ListPrice
 FROM       Production.Product;
SET NOCOUNT OFF
GO
```

**pr_UrunleriGetir** isimli sproc'u **WITH RECOMPILE** ile her çalıştırılmada yeniden derlenecek şekilde düzenledik.

Ancak SQL Server'da sproc'ların çok önemli bir performans özelliği var. Bu özellik, sproc'un bir kez derlenmesi ve sonrasında, ilk derlemede çalıştırma planı (*execution plan*) oluşturulduğu için, sonraki çalıştırılmalarda daha hızlı çalışarak

performansı artırıyordu. Ancak **WITH RECOMPILE** kullanmamız ile birlikte, artık prosedür hafızasında bu sproc için bir çalıştırma planı tutulmayacaktır.

Sproc'un her çalıştırılmasında yeniden derlemek yerine, sadece istediğimiz zaman derleyerek, sproc'un yeni yapıya göre yeniden derlenmesi ve yeni bir çalıştırma planına sahip olmasını sağlayabiliriz.

Bu işlem için;

```
EXECUTE prosedur_ismi WITH RECOMPILE
```

Bir başka yöntem ise, sproc'un bir sonraki çalıştırılmada, prosedürün derlenerek çalıştırılmasını sağlayabiliriz. Bu yöntem ile, prosedür hafızasındaki çalıştırma planı bir defaya mahsus silinerek yeni çalıştırma planı oluşturulur. Bu işlem için sp_recompile sistem prosedürü kullanılır.

```
EXECUTE sp_recompile prosedur_ismi | tablo_ismi
```

- **prosedur_ismi:** Sadece bir prosedürün takip eden ilk çalıştırılmasında yeniden derlemeye zorlamak için verilir.
- **tablo_ismi:** Bir tabloya erişen tüm prosedürleri, bir sonraki seferde yeniden derlemeye zorlamak için verilir.

## Stored Procedure'ler İçin İzinleri Yönetmek

Bir prosedür oluşturulduktan sonra kullanıcının sproc'u kullanabilmesi için izin verilmesi gerekir.

Bir sproc'u public role kapatmak.

```
DENY ON prosedur_ismi TO public
```

Bir kullanıcıya sproc'a erişim izni vermek için;

```
GRANT ON prosedure_ismi TO kullanici_ismi
```

# Stored Procedurelerde Parametre Kullanimi

Şu ana kadar prosedürlerin parametresiz kullanımını gördük. Ancak prosedürler hem dışarıdan parametre alabilir, hem de içerideki işlem sonucunda oluşacak bir değeri dışarıya parametre olarak gönderebilir. Bir sproc, en fazla 1024 parametre alabilir. Prosedürden dışarıya gönderilen parametreye **döndürülen parametre** (*return parameter*) denir.

## Girdi Parametreler (Input Parameters)

Sproc'un işlevselliğini ve programsal yeteneklerini artırmak için kullanılan özelliklerin başında, girdi parametreleri gelir. Girdi parametre kullanımı ile, dışarıdan bir değer sproc'a parametre olarak gönderilir ve sproc içerisinde programsal olarak kullanılır.

Girdi parametresi kullanan bir sproc oluşturalım. Sproc, dışarıdan bir ürün numarası alarak **`Production.Product`** tablosunda aratsın ve sonucunda ilgili koşula sahip kayıt ya da kayıtları listelesin.

```
CREATE PROCEDURE pr_UrunAra(@ProductNumber NVARCHAR(25))
AS
SET NOCOUNT ON
IF @ProductNumber IS NOT NULL
 SELECT
  ProductID, ProductNumber,
  Name, ListPrice
 FROM
  Production.Product
 WHERE
  ProductNumber = @ProductNumber;
SET NOCOUNT OFF
```

Sorgunun çalışmasıyla birlikte artık ürün arayabileceğimiz bir sproc'a sahip olduk.

## Girdi Parametreler İle Stored Procedure Çağırmak

Dışarıdan parametre alan prosedür iki farklı şekilde kullanılabilir.

**Söz Dizimi:**

```
EXEC prosedur_ismi @parametre_ismi = deger
```

ya da

```
EXEC prosedur_ismi parametre_degeri
```

Şimdi oluşturduğumuz **pr_UrunAra** prosedürünü çalıştıralım.

**1. Yöntem**

```
EXEC pr_UrunAra @ProductNumber = 'SA-T872';
```

**2. Yöntem**

```
EXEC pr_UrunAra 'SA-T872';
```

| | ProductID | ProductNumber | Name | ListPrice |
|---|---|---|---|---|
| 1 | 522 | SA-T872 | HL Touring Seat Assembly | 196,92 |

İki yöntemi de kullanarak girdi parametreli Stored Procedür'ü kullandık ve başarılı bir şekilde sorgu sonucunu aldık.

## Tablo Tipi Parametre Alan Stored Procedure'ler

SQL Server, Stored Procedure'e tablo tipi değişken gönderme özelliğine sahiptir. Bu özellik, SQL Server 2008 ile birlikte geliştirilmiştir.

Bu özelliği hemen uygulama üzerinde inceleyelim.

**AdventureWorks** veritabanında kategorileri ve kategorilere bağlı alt kategorileri listeleyelim.

- İlk olarak girdi parametre olarak kullanılacak kullanıcı tanımlı tablo tipini oluşturalım:

```
CREATE TYPE dbo.ProductType AS TABLE
(
 ProductCategoryID INT,
 Name       VARCHAR(40)
);
```

- Sonrasında, bu tipi parametre olarak alacak Stored Procedure'ü oluşturalım.

```
CREATE PROCEDURE pr_AllCateogries
(
 @productCategory AS dbo.ProductType READONLY
)
AS
BEGIN
SET NOCOUNT ON
 SELECT
 PC.Name AS Category,
 PS.Name AS SubCategory
 FROM Production.ProductSubcategory AS PS
 JOIN @productCategory AS PC ON
 PC.ProductCategoryID = PS.ProductCategoryID;
SET NOCOUNT OFF
END;
```

- Son olarak, tablolarımızdaki 4 kategori ile doldurduğumuz **`ProductType`** türündeki tablo tipi değişkeni Stored Procedure'e göndererek test edelim.

```
DECLARE @category AS dbo.ProductType;
INSERT INTO @category (ProductCategoryID, Name)
SELECT TOP(4) ProductCategoryID, Name FROM Production.ProductCategory;
EXEC pr_AllCateogries @category;
```

| | Category | SubCategory |
|---|---|---|
| 1 | Bikes | Mountain Bikes |
| 2 | Bikes | Road Bikes |
| 3 | Bikes | Touring Bikes |
| 4 | Components | Handlebars |
| 5 | Components | Bottom Brackets |
| 6 | Components | Brakes |
| 7 | Components | Chains |

## Çıkış Parametrelerle Çalışmak (Output Parameters)

Çıktı parametreleri, **OUTPUT** parametre (*output parameter*) olarak bilinir. Kendisine gönderilen parametreyi, içerisindeki işlem doğrultusunda değer ile doldurarak dışarıya gönderir. Bu işlem, içeriden dışarıya doğru gerçekleşir. Prosedür, kendisini çağıran başka bir prosedüre ya da dışarıya gönderilen değeri alacak başka bir alıcıya, kendi ürettiği değeri gönderir.

Çıktı parametreleri için detaylı bir örnek yapalım.

Stored Procedure'den istenen özellikler:

- 2 sayı alarak matematiksel hesaplama yapabilmeli.
- Hangi matematiksel işlemi yapacağına kullanıcı karar vermeli.
- Hesaplama işlemi sonucunu çıktı parametre olarak vermeli.
- Dışarıdan **NULL** değer gönderilirse işlem yapmamalı.
- 4 işlem haricinde girdi parametresi gönderilirse, geriye 0 (*sıfır*) hata kodunu göndermeli.

Bu işlemi gerçekleştirmek için;

```
CREATE PROC pr_HesapMakinesi
(
 @sayi1 INT,
 @sayi2 INT,
 @islem SMALLINT,
 @sonuc INT OUTPUT
)
AS
SET NOCOUNT ON
IF @islem IS NOT NULL
 IF(@islem = 0)
  SELECT @sonuc = (@sayi1 + @sayi2);
 ELSE IF(@islem = 1)
  SELECT @sonuc = (@sayi1 - @sayi2);
 ELSE IF(@islem = 2)
  SELECT @sonuc = (@sayi1 * @sayi2);
 ELSE IF(@islem = 3)
```

```
  SELECT @sonuc = (@sayi1 / @sayi2);
 ELSE
  SELECT @sonuc = (0);
SET NOCOUNT OFF
```

## Çıkış Parametrelerini Almak

Çıkış parametrelerini alabilmek için şu yol izlenmeli.

- Prosedürün ürettiği değeri alabilmek için, döndüreceği değişkenin veri tipine uygun değişken tanımlanır.
- Tanımlanan bu değişkenler, prosedüre `OUTPUT` parametre olarak verilir.
- Prosedür çalıştırıldığında, çıkış parametrelerindeki değer bu tanımlanan değişkenlerden okunur.

Hazırladığımız uygulamanın çıkış parametre değerini alalım.

Prosedürümüz 4 parametreden oluşuyor.

- **1. parametre:** Hesaplama yapılacak ilk değer.
- **2. parametre:** Hesaplama yapılacak ikinci değer.
- **3. parametre:** Hangi hesaplama işleminin yapılacağı (`0`: toplama, `1`: çıkarma, `2`: çarpma, `3`: bölme)
- **4. parametre:** Çıkış parametresi. (`@sonuc`)

```
-- Çıkış parametresinden alınan değerin tutulacağı değişken.
DECLARE @sonuc INT;

-- Prosedürü çalıştırmak için gönderilen parametreler.
EXEC pr_HesapMakinesi 7,6,2,@sonuc OUT;

-- Çıkış parametresinden alınan değerin gösterilmesi.
SELECT @sonuc;
```

Çıkış parametrelerini bir başka kullanımı şu şekildedir.

```
DECLARE @sonucOut INT;
EXEC pr_HesapMakinesi @sayi1=7, @sayi2=6, @islem = 2, @sonuc = @
sonucOut OUT;
SELECT 'Çarpım', @sonucOut;
GO
```

| | (No column name) | (No column name) |
|---|---|---|
| 1 | Çarpım | 42 |

# RETURN Deyimi

Çıkış parametrelerini kullanmaya gerek kalmadan Stored Procedure içerisinden değer almayı sağlar.

- 0 ile -99 arasındaki değerler, sistem bazı durumları belirlemek üzere ayrılmıştır.
- Bu kodların ilk 16 tanesinin SQL Server'da özel önemi vardır.
- -15 ile -99 arasındaki değerler, daha sonradan kullanılmak üzere ayrılmıştır.

Örneğin, 0 değeri, prosedürün çalışmasında bir hata oluşmadığını belirtir. Bu gibi sistem tarafından ayrılmış (0 ve -99 arasındaki) değerler haricinde geri değer döndürülebilir.

Hesaplama işlemleri için kullandığımız **pr_HesapMakinesi** prosedürünü değiştirerek, **RETURN** ile değer döndürebilir şekilde oluşturalım. Daha önce var olan bu prosedürü **ALTER** ile değiştiriyoruz.

```
ALTER PROC pr_HesapMakinesi
(
 @sayi1 INT,
 @sayi2 INT,
 @islem SMALLINT,
 @sonuc INT OUTPUT
)
AS
SET NOCOUNT ON
IF @islem IS NOT NULL
 IF(@islem = 0)
  SELECT @sonuc = (@sayi1 + @sayi2);
 ELSE IF(@islem = 1)
  SELECT @sonuc = (@sayi1 - @sayi2);
 ELSE IF(@islem = 2)
  SELECT @sonuc = (@sayi1 * @sayi2);
 ELSE IF(@islem = 3)
  SELECT @sonuc = (@sayi1 / @sayi2);
 ELSE
  SELECT @sonuc = (0);
 RETURN(@sayi1 + @sayi2);  -- Eklenen RETURN deyimi
SET NOCOUNT OFF
```

Aşağıdaki şekilde çağırabiliriz.

```
DECLARE @sonucOut INT,
 @toplam INT;
EXEC @toplam = pr_HesapMakinesi @sayi1=7,
                                @sayi2=6,
                                @islem = 2,
                                @sonuc = @sonucOut OUT;
SELECT 'Çarpım', @sonucOut,
 'Return ile Toplam : ', @toplam;
```

| | (No column name) | (No column name) | (No column name) | (No column name) |
|---|---|---|---|---|
| 1 | Çarpim | 42 | Retum ile Toplam : | 13 |

# EXECUTE AS Modül Çalıştırma Bağlamları

Oluşturulan prosedürlerin, kullanıcı hakları arasındaki farklılıklar nedeni ile başka bir kullanıcının haklarını kullanabilmesini sağlar.

```
{ EXEC  |  EXECUTE } AS { CALLER  |  SELF  |  OWNER  |  'kullanici' }
```

**EXECUTE AS**, izne bağlı olması gerekmeyen ifadeleri izne bağlamak için kullanılır. Örneğin; tablo içerisindeki verileri silmek için kullanılan **TRUNCATE TABLE** ifadesini bir izne bağlayabilirsiniz.

Bu izne bağlama işlemi için, izne bağlamak istediğiniz ilgili ifadeyi, bir Stored Procedure içerisine alarak, bu ifadeyi kullanmalarına izin vereceğiniz kullanıcılara Stored Procedure'ü kullanım izni vererek erişmelerini sağlayabilirsiniz.

## EXECUTE AS CALLER

Oluşturulan Stored Procedure'ü, kendisini çağıran kullanıcı adına çalıştıracağını belirtir. Yani, SQL Server'da normal Stored Procedure kullanımı ile aynı işlemi gerçekleştirir.

Belirtilen ID değerine sahip ürünü getiren bir prosedür oluşturalım.

```
CREATE PROC pr_UrunGetir
(
 @ProductID INT
)
WITH EXECUTE AS CALLER
AS
SELECT
 ProductID, Name, ProductNumber
FROM
 Production.Product
WHERE
 ProductID = @ProductID;
```

SQL Server'da varsayılan kullanımın **EXECUTE AS CALLER** olduğunu söylemiştik. Prosedüre, aşağıdaki ifadeyi eklemeseydik de aynı şekilde çalışacaktı.

```
WITH EXECUTE AS CALLER
```

## EXECUTE AS 'KULLANICI'

Prosedürü çağıran kişi dışında, başka bir kullanıcı hesabı ile çalışmasını sağlar.

```
CREATE PROC pr_UrunGetir
(
 @ProductID INT
)
WITH EXECUTE AS kullanici_ismi
AS
SELECT
 ProductID, Name, ProductNumber
FROM
 Production.Product
WHERE
 ProductID = @ProductID;
```

Herhangi bir kullanıcı, **pr_UrunGetir** Stored Procedure'ünü çalıştırmak isterse, SQL Server otomatik olarak belirtilen kullanıcı ismine yönlenecek ve bu kullanıcının yetkilerini kullanmaya başlayacaktır. Bir başka deyişle, Stored Procedure için bir yönlendirmeli yetkilendirme yapılmaktadır.

### EXECUTE AS SELF

**EXECUTE AS**'i tanımlayan ya da değiştiren kullanıcıyı vermek gerektiğinde kullanılır. Sahiplik zinciri değişse bile bu durum değişmez.

### EXECUTE AS OWNER

Prosedürün içerdiği kodlar prosedür sahibi adına çalıştırılır. Eğer bir sahibi yok ise, prosedürün içinde bulunduğu şemanın sahibi adına çalışır.

Modülün içeriğini değiştirmeden, kimin adına çalıştığını değiştirmek istiyorsanız, çalışma zamanında otomatik olarak sahip, adına çalıştığı kullanıcı olacaktır.

## WITH RESULT SETS İLE STORED PROCEDURE ÇAĞIRMAK

Stored Procedure'lerin bazen, tablolardan aldıkları verileri farklı isim ya da veri tipinde getirmelerini isteyebiliriz. Bu gibi durumlarda **WITH RESULT SET** yan cümlesini kullanarak Stored Procedure çağırabiliriz.

Daha önce oluşturduğumuz **pr_UrunGetir** prosedüründe **WITH RESULT SETS**'i kullanalım. Daha önce oluşturduysanız **CREATE** ile tekrar oluşturmanıza gerek yoktur.

```
CREATE PROC pr_UrunGetir
(
 @ProductID INT
)
AS
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID = @ProductID;
```

**WITH RESULT SETS** ile prosedürü çağıralım.

```
EXEC pr_UrunGetir 4
WITH RESULT SETS(
 (
  KOD       VARCHAR(20),
```

```
  [Ürün Adı] VARCHAR(20),
  [Ürün Numarası] VARCHAR(7)
 )
);
```

| | KOD | Ürün Adı | Ürün Numarası |
|---|---|---|---|
| 1 | 4 | Headset Ball Bearing | BE-2908 |

# Stored Procedure Güvenliği

Stored procedure'ler SQL Server'ın programsal yeteneklerini geliştirdikleri gibi, kullanımına özen gösterilmesinin başlıca sebeplerinden biri de veri güvenliğidir. Uygulama tarafından veritabanına direk erişimi engelleyecek bir katman olarak da düşünebiliriz. Stored Procedure'lerin kullanıldığı bir projede, kullanıcı tarafından ne tür istek ya da saldırı kodu gelirse gelsin, T-SQL yeteneklerinizi doğrultusunda Stored Procedure içerisinde bu sorguları düzenleyebilir ve filtreleyebilirsiniz.

Stored Procedure'ü, uygulamanızdan, başka girişi olmayan bir veritabanına açılan kapı olarak düşünebilirsiniz. Bu kapıyı ne kadar güvenli geliştirirseniz verileriniz o kadar güvenli bir veritabanında saklanacaktır.

Stored Procedure'ler veritabanındaki verilere erişimi güvenli hale getirmek için kullanıldığı gibi, kendi verilerini de koruyabilecek yeteneklere sahiptir. Prosedürlerin kendi kaynak kodlarını şifreleyerek gizleyebilmeleri için, prosedür oluştururken ya da **ALTER** ile değiştirirken **WITH ENCRYPTION** ifadesini kullanabilirsiniz.

# Stored Procedure'lerin Şifrelenmesi

SQL Server için geliştirilen uygulamalar çoğunlukla lisanslı ya da benzeri modellerde ticari amaçlı olarak satılmakta ve satın alanlar tarafından kendi bilgisayarlarında kullanılmaktadır. Tabi ki, bu yazılımların başka bilgisayarlarda çalışıyor olması, geliştirici firma açısından bir risktir. Yazılımın kurulduğu bilgisayarda bu yazılımın veritabanı algoritması ya da çalışma modeli çalınabilir, izinsiz erişimler için araştırmalar yapılabilir.

Bir Stored Procedure'e ait kaynak kodların görüntülenmesini engellemek için, **WITH ENCRYPTION** ile şifreleme özelliğini kullanılmalıdır.

**pr_UrunGetir** Stored Procedure'ünün içeriğini **sp_helptext** sistem prosedürü ile görüntüleyelim.

```
sp_helptext 'pr_UrunGetir';
```

| | Text |
|---|---|
| 1 | |
| 2 | CREATE PROC pr_UrunGetir |
| 3 | ( |
| 4 | @ProductID INT |
| 5 | ) |
| 6 | AS |
| 7 | SELECT ProductID, Name, ProductNumber |
| 8 | FROM Production.Product |
| 9 | WHERE ProductID = @ProductID; |

Bu kullanım ile **pr_UrunGetir** prosedürünün içeriğini görüntüledik.

Şimdi prosedür içeriğini şifreleyelim.

```
ALTER PROC pr_UrunGetir
(
 @ProductID INT
)
WITH ENCRYPTION
AS
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID = @ProductID;
```

**sp_helptext** ile prosedür içeriğini tekrar görüntülemeye çalıştığımızda aşağıdaki sonuç ile karşılaşırız.

```
sp_helptext 'pr_UrunGetir';
```

```
The text for object 'pr_UrunGetir' is encrypted.
```

Prosedürümüzün içeriği şifrelendi. Artık prosedürümüzün içeriği erişilemez oldu.

Stored Procedure'leri şifrelemek etkili bir güvenlik yöntemi olsa da, beraberinde güvenlik riskleri de getirir. Bu nedenle, prosedürler şifrelenmeden

önce mutlaka içerikleri yedeklenmeli, hatta olası yedek kayıplarına karşı da algoritma koruma altına alınmalıdır.

Ancak şifrelenmiş bir prosedürün içeriği görüntülenmeli ise;

- SQL Server Dedicated Admin Connection (DAC) kullanarak SQL Server'a bağlanılır ve **`sys.sysobjvalues`** sistem view'i kullanılmalı.
- SQL Server'da bu işi yapan 3. parti uygulamalar ile bu işlem gerçekleştirilmeli.

# Stored Procedure'ler Hakkında Bilgi Almak

SQL Server'da oluşturduğumuz Stored Procedure nesnelerinin takibi ve yönetimi için Microsoft, bazı sistem prosedürleri ve view'leri oluşturmuştur. Bu view ve prosedürleri kullanarak kendi oluşturduğumuz prosedürler hakkında bilgi alabiliriz.

**`Sys.Procedures`** kullanarak prosedürler hakkındaki bilgileri listeleyelim.

```
SELECT
 Name, Type, Type_Desc,
 Create_Date, Modify_Date
FROM
 Sys.Procedures;
```

| | Name | Type | Type_Desc | Create_Date | Modify_Date |
|---|---|---|---|---|---|
| 1 | uspGetBillOfMaterials | P | SQL_STORED_PROCEDURE | 2012-03-14 13:14:55.710 | 2012-03-14 13:14:55.710 |
| 2 | uspGetEmployeeManagers | P | SQL_STORED_PROCEDURE | 2012-03-14 13:14:55.723 | 2012-03-14 13:14:55.723 |
| 3 | uspGetManagerEmployees | P | SQL_STORED_PROCEDURE | 2012-03-14 13:14:55.750 | 2012-03-14 13:14:55.750 |
| 4 | uspGetWhereUsedProductID | P | SQL_STORED_PROCEDURE | 2012-03-14 13:14:55.757 | 2012-03-14 13:14:55.757 |
| 5 | uspUpdateEmployeeHireInfo | P | SQL_STORED_PROCEDURE | 2012-03-14 13:14:55.760 | 2012-03-14 13:14:55.760 |

Sorgu ekranının oturumu hangi veritabanını kullanıyorsa, o veritabanına ait prosedürler hakkındaki bilgiler listelenecektir.

Seçili veritabanını değiştirmek için;

- **SSMS** içerisinde, **Available Databases** seçim menüsü kullanılabilir.
- Sorgu ekranında aşağıdaki gibi bir veritabanı seçim sorgusu yazabilirsiniz.

```
USE db_ismi
GO
SELECT ...
```

Tam olarak istediğimiz prosedür ya da prosedürleri belirtmek için isimlendirmelerimize göre arama da yapabiliriz.

Örneğin; **pr_** ile başlayan birden fazla prosedürümüzü aşağıdaki şekilde listeleyebiliriz.

```
SELECT
 Name, Type, Type_Desc,
 Create_Date, Modify_Date
FROM
 Sys.Procedures
WHERE
 Name LIKE 'pr_%';
```

| | Name | Type | Type_Desc | Create_Date | Modify_Date |
|---|---|---|---|---|---|
| 1 | pr$UrunleriGetir | P | SQL_STORED_PROCEDURE | 2013-01-11 22:25:02.143 | 2013-01-11 22:33:47.253 |
| 2 | pr_UrunAra | P | SQL_STORED_PROCEDURE | 2013-01-11 23:38:53.413 | 2013-01-12 00:22:36.380 |
| 3 | pr_AllCateogries | P | SQL_STORED_PROCEDURE | 2013-01-12 01:05:02.673 | 2013-01-12 01:07:19.663 |
| 4 | pr_HesapMakinesi | P | SQL_STORED_PROCEDURE | 2013-01-12 01:21:15.880 | 2013-01-12 09:53:43.277 |
| 5 | pr_UrunGetir | P | SQL_STORED_PROCEDURE | 2013-01-12 10:57:24.807 | 2013-01-12 21:32:04.483 |
| 6 | pr_UrunleriGetir | P | SQL_STORED_PROCEDURE | 2013-01-12 21:44:16.727 | 2013-01-12 21:44:16.727 |

**sys.sql_modules** kullanarak prosedürler hakkındaki bilgileri listeleyelim.

```
SELECT * FROM Sys.Sql_Modules;
```

Farklı sistem yapıları ile birlikte, sorgu içerisinde kullanarak, gelişmiş sorgular oluşturulabilir.

```
SELECT  Definition, O.Object_ID, Create_Date,
        OBJECT_NAME(O.Object_ID) Prosedur_Ismi
        FROM Sys.Sql_Modules M
        INNER JOIN Sys.Objects O ON
        M.Object_ID = O.Object_ID
        WHERE O.type = 'P';
```

| | Definition | Object_ID | Create_Date | Prosedur_Ismi |
|---|---|---|---|---|
| 1 | CREATE PROCEDURE [dbo].[uspGetBillOfMaterials] @... | 231671873 | 2012-03-14 13:14:55.710 | uspGetBillOfMaterials |
| 2 | CREATE PROCEDURE [dbo].[uspGetEmployeeManagers] ... | 247671930 | 2012-03-14 13:14:55.723 | uspGetEmployeeManagers |
| 3 | CREATE PROCEDURE [dbo].[uspGetManagerEmployees] ... | 263671987 | 2012-03-14 13:14:55.750 | uspGetManagerEmployees |
| 4 | CREATE PROCEDURE [dbo].[uspGetWhereUsedProductI... | 279672044 | 2012-03-14 13:14:55.757 | uspGetWhereUsedProductID |
| 5 | CREATE PROCEDURE [HumanResources].[uspUpdateEm... | 295672101 | 2012-03-14 13:14:55.760 | uspUpdateEmployeeHireInfo |
| 6 | CREATE PROCEDURE [HumanResources].[uspUpdateEm... | 311672158 | 2012-03-14 13:14:55.763 | uspUpdateEmployeeLogin |

# Stored Procedure'lerin Kaldırılması

Tüm nesnelerde olduğu gibi sproc silmek oldukça kolaydır.

```
DROP PROC|PROCEDURE sproc_ismi
```

Bu sorgu yapısı ile birlikte ismi belirtilen sproc ortadan kalkar.