İLERİ SEVİYE SORGULAMA

5

Veritabanı yeteneklerinden tam anlamıyla yararlanabilmek için geliştirilen bazı ek özellikler mevcuttur. Bu özellikler tek bir kategori altında değil, birçok farklı kategoride SQL Server'a ileri seviye yetenek kazandırmak için geliştirilmiştir.

Bu bölümde, veri sorgulama, veri yönetimi, gruplama ve daha birçok ileri seviye sorgulama konularına değineceğiz.

Alt Sorgu Nedir?

Alt sorgu (*SubQuery*), parantezler kullanılarak başka bir sorgu içine yerleştirilmiş T-SQL sorgusudur. Bir sorgu sonucunda dönen veriyi, başka bir sorgu içerisinde kullanarak sorgularda kullanmaya yarar.

Alt sorgular genellikle birkaç amaç ile kullanılır.

- Üst sorgudaki her bir kayıt için arama yapılmasını sağlamak.
- Bir sorguyu birkaç mantıksal basamağa bölmek.
- **`IN`**, **`ANY`**, **`ALL`**, **`EXISTS`** özellikleri ile **`WHERE`** koşulu için liste sağlamak.

Alt sorgular yapmak istediğiniz işleme göre, zor ya da kolaydır diyebiliriz. Temel alt sorgu kullanımı oldukça kolaydır. Ancak karmaşık işlemlerin gerçekleştirileceği alt sorgular içinde barındıracağı SQL sorgularının karmaşıklığı nedeniyle alt sorguyu karmaşık ve geliştirilmesini zorlaştırabilir.

Alt sorguların alt (iç sorgu) ve üst (dış sorgu) sorgulara sahip olması, iki farklı sorgu ile uğraşmayı ve bu sorguların birbirleriyle kullanılabilmesini sağlayacak şekilde iyileştirilmesini gerektirir.

Alt sorgular, **JOIN**'ler ile benzer amaçlar için kullanılır. Performans açısından iki özelliğin de kullanılması gereken farklı zamanlar olabilir. Genellikle alt sorgu ve **JOIN** özellikleri arasında tercih yapanlar kendi kullandıkları yöntemlerin (alt sorgu ya da **JOIN**) daha performanslı ve doğru yol olacağını düşünür. Ancak araştırma ve tecrübelerime dayanarak söyleyebilirim ki, hiç bir zaman en doğru yol bu özelliklerden birisi değildir. Bazen doğru çözüm, alt sorgular olabileceği gibi, bazen de **JOIN**'ler olabilmektedir. Bu konuda kendi fikir ve tecrübelerinizi edinmeniz için iki özelliğin detaylarını da incelemeye çalışacağız.

İç İçe Alt Sorgular Oluşturmak

İç içe alt sorgular tek bir yönde ilerler. Tek değer ya da bir liste değer döndürebilir. Tek değer döndürmek için genel olarak **WHERE** koşulunda eşittir (**=**) ile bildirim yapılır. Liste değer döndürülmek isteniyor ise, **IN** kullanılabilir.

Söz Dizimi (Tek satır döndüren):

```
SELECT sutun_ismi1 [, sutun_ismi2, ...]
FROM tablo_ismi
WHERE sutun_ismi1 = (
        SELECT sutun_ismi1
        FROM tablo_ismi
        WHERE tek_satir_donduren_kosul)
```

ya da

Söz Dizimi(Liste döndüren):

```
SELECT sutun_ismi1 [, sutun_ismi2, ...]
FROM tablo_ismi
WHERE sutun_ismi1 IN (SELECT sutun_ismi1
          FROM tablo_ismi
          WHERE kosul)
```

Tekil Değerler Döndüren İç İçe Sorgular

Tekil değer döndüren alt sorgu oluşturmak pratikte standart **WHERE** koşulu kullanmaya benzer. Ancak alt sorgular, daha karmaşık ve dinamik sorgular üzerinde bu işlemi yapabilmemizi sağlar.

Alt sorguların neden kullanıldığını kavrayabilmek için bir sorun ve çözüm uygulaması yapalım.

Yoğun satış işlemi gerçekleşen bir firmada, yıllar sonra satılan ilk ürünün, ürün satış tarihi ve ilk satılan ürünün **ProductID** değeri öğrenmek istenebilir.

Bu durumda **JOIN** kullanılması gerektiğini düşünebiliriz. Ancak tek başına **JOIN** yeterli olur mu inceleyelim.

Satışı yapılmış olan ilk ürünün tarihini biliyorsak;

```
SELECT
DISTINCT SOH.OrderDate AS SiparisTarih,
SOD.ProductID AS UrunNO
FROM Sales.SalesOrderHeader SOH
JOIN Sales.SalesOrderDetail SOD
ON   SOH.SalesOrderID = SOD.SalesOrderID
WHERE OrderDate = '07/01/2005';
```

	SiparisTarih	UrunNO
1	2005-07-01 00:00:00.000	707
2	2005-07-01 00:00:00.000	708
3	2005-07-01 00:00:00.000	709
4	2005-07-01 00:00:00.000	710
5	2005-07-01 00:00:00.000	711
6	2005-07-01 00:00:00.000	712
7	2005-07-01 00:00:00.000	714

Kullandığım **AdventureWorks2012** veritabanında ilk satış tarihi `07/01/2005`'dir.

Bu sorgu sonucunda listelenen kayıtlarda, saat zaman biriminin tüm kayıtlarda 0 (*sıfır*) olarak belirtilmesinden dolayı, aynı günde yapılan tüm satışları ilk satış olarak göstermektedir. Ancak gerçek uygulamalarda saniye ve salise değerlerinin bile aynı olduğu satış ve sipariş olma olasılığı çok düşüktür.

Evet, istediğimiz sonucu aldık. Ancak dikkat ederseniz, burada dinamik bir programlama yapmadık. İlk sipariş tarihini biliyorduk ve **WHERE** koşuluna bu değeri vererek filtreleme gerçekleştirdik. Genel kullanım bu şekilde değildir. Büyük veritabanı ve uygulamalarda her şey dinamik olmalıdır.

Oluşturduğumuz bu sorguyu kullandığımızı ve daha sonra bu ilk satılan ürünün veritabanından silindiğini düşünelim. Bu durumda sorgumuz hata verecektir. Ancak dinamik bir sorgu oluştursaydık bu hatayı almazdık. İlk satılan ürün silinse bile ondan sonra satılan ilk ürün dinamik olarak veritabanından alınabilir ve sorgu içerisinde kullanılabilirdi.

Aynı işlemi dinamik olarak nasıl yapabileceğimize bakalım.

```
DECLARE @IlkSiparisTarih SMALLDATETIME;
SELECT  @IlkSiparisTarih = MIN(OrderDate)
        FROM Sales.SalesOrderHeader;
SELECT
        DISTINCT SOH.OrderDate AS SiparisTarih,
        SOD.ProductID AS UrunNO
FROM Sales.SalesOrderHeader SOH
JOIN Sales.SalesOrderDetail SOD
ON   SOH.SalesOrderID = SOD.SalesOrderID
WHERE OrderDate = @IlkSiparisTarih;
```

	SiparisTarih	UrunNO
1	2005-07-01 00:00:00.000	707
2	2005-07-01 00:00:00.000	708
3	2005-07-01 00:00:00.000	709
4	2005-07-01 00:00:00.000	710
5	2005-07-01 00:00:00.000	711
6	2005-07-01 00:00:00.000	712
7	2005-07-01 00:00:00.000	714

İlk örnekte dinamik olmayan sorgumuzu ikinci örneğimizde dinamik hale getirdik. Ancak kod yazımını azaltacak daha farklı bir yöntem var.

Alt sorgu kullanarak bu işlemi daha az kod ile gerçekleştirebiliriz. Daha az kod, her zaman daha kullanışlı ve anlaşılabilir bir uygulama yöntemi olacaktır.

```
SELECT
 DISTINCT SOH.OrderDate AS SiparisTarih,
 SOD.ProductID AS UrunNO
FROM Sales.SalesOrderHeader SOH
JOIN Sales.SalesOrderDetail SOD
ON   SOH.SalesOrderID = SOD.SalesOrderID
WHERE SOH.OrderDate = (SELECT MIN(OrderDate)
                       FROM Sales.SalesOrderHeader);
```

	SiparisTarih	UrunNO
1	2005-07-01 00:00:00.000	707
2	2005-07-01 00:00:00.000	708
3	2005-07-01 00:00:00.000	709
4	2005-07-01 00:00:00.000	710
5	2005-07-01 00:00:00.000	711
6	2005-07-01 00:00:00.000	712
7	2005-07-01 00:00:00.000	714

JOIN ifadesi ve **WHERE** koşulu içeren sorgumuza bir alt sorgu ekleyerek istediğimiz sonucu listeledik. Parantezler içerisindeki alt sorguda tek yaptığımız **Sales.SalesOrderHeader** tablosundaki **OrderDate** sütunundaki en küçük değeri **MIN** fonksiyonu ile belirlemek oldu.

Çoklu Sonuç Döndüren İç İçe Sorgular

Alt sorguların tercih edildiği ve bu sorguların en çok kullanım alanına sahip olduğu özelliği, alt sorguların çoklu sonuç döndürdüğü durumlardır.

ProductCategoryID değeri **1**, **2** ve **3** değerine sahip olan kategorilere ait alt kategorileri listeleyelim.

```
SELECT PC.Name, PSC.Name
FROM Production.ProductCategory AS PC
JOIN Production.ProductSubCategory AS PSC
ON PC.ProductCategoryID = PSC.ProductCategoryID
WHERE PC.ProductCategoryID IN (1,2,3);
```

	Name	Name
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

Aynı sorguyu farklı bir şekilde de gerçekleştirebilirdik.

```
SELECT PC.Name, PSC.Name
FROM Production.ProductCategory AS PC
JOIN Production.ProductSubCategory AS PSC
ON PC.ProductCategoryID = PSC.ProductCategoryID
WHERE PC.ProductCategoryID IN(
                              SELECT ProductCategoryID
                              FROM Production.ProductCategory
                              WHERE ProductCategoryID = 1
                              OR ProductCategoryID = 2
                              OR ProductCategoryID = 3);
```

Oluşturduğumuz bu sorgu önceki **IN** kullanımı ile aynı sonucu üretti. Ancak fark ettiğiniz gibi daha fazla kod yazdık ve sorgumuz daha karmaşık hale geldi. Bu tür durumlarda basit **IN**(1,2,3) kullanımı tercih edilmelidir.

Firmamızdaki bir birim için daha önce iş başvurusu yapan kişiler arasından bazı adaylar ile görüşülecek. Bu adayların listesini oluşturmak için veritabanımızda bir sorgu hazırlamalıyız. Bu iş için bize yardımcı olacak tablolar şunlardır;

- **Person.Person**
- **Person.Phone**
- **HumanResources.JobCandidate**

Çalışan adaylarının kayıtlarını listeleyelim.

```
SELECT
 PP.BusinessEntityID,
 PP.FirstName,
 PP.LastName,
 P.PhoneNumber
FROM Person.Person PP
JOIN Person.PersonPhone AS P
ON PP.BusinessEntityID = P.BusinessEntityID
WHERE P.BusinessEntityID IN (
    SELECT DISTINCT BusinessEntityID
    FROM HumanResources.JobCandidate
    WHERE BusinessEntityID IS NOT NULL);
```

	BusinessEntityID	FirstName	LastName	PhoneNumber
1	212	Peng	Wu	164-555-0164
2	274	Stephen	Jiang	238-555-0197

Aslında **HumanResources.JobCandidate** tablomuzda, iş başvurusu yapmış daha fazla kişinin kaydı var. Ancak biz sorgumuzu hazırlarken **BusinessEntityID** değeri **NULL** olmayan kayıtların listelenmesini istediğimiz için 2 kayıt listelendi. **BusinessEntityID** değeri **NULL** olan kayıtlar üzerinden bir ilişki kurulamayacağı için o kayıtları listelememize gerek kalmadı.

Aynı işlemi sadece **JOIN** kullanarak da gerçekleştirebiliriz.

```
SELECT
 PP.BusinessEntityID,
 PP.FirstName,
 PP.LastName,
 P.PhoneNumber
FROM Person.Person PP
JOIN Person.PersonPhone AS P
ON PP.BusinessEntityID = P.BusinessEntityID
JOIN HumanResources.JobCandidate AS JC
ON P.BusinessEntityID = JC.BusinessEntityID
WHERE JC.BusinessEntityID IS NOT NULL;
```

	BusinessEntityID	FirstName	LastName	PhoneNumber
1	212	Peng	Wu	164-555-0164
2	274	Stephen	Jiang	238-555-0197

Alt sorgunun gerektiği durumlar elbette vardır. Ancak **JOIN** ile gerçekleştirebileceğiniz sorgularda **JOIN**'i tercih etmenizi öneririm. Performans açısından bazı durumlar haricinde **JOIN** daha etkilidir.

Türetilmiş Tablolar

Türetilmiş tablolar, alt sorguların özel bir halidir. Bir sorgudan gelen kayıtları, tabloymuş gibi kullanmak için tercih edilir.

Türetilmiş tablolar, **FROM** ifadesinden sonra gelen **SELECT** ifadesinin parantez içerisine alınıp bir takma ad verilmesiyle oluşturulur.

Söz Dizimi:

```
SELECT İfadesi
FROM (SELECT İfadesi) [AS]
turetilmis_tablo_takma_ismi [(sutun_takma_ismi, ...]
```

En fazla alt kategoriye sahip kategoride kaç alt kategori olduğunu öğrenmek için;

```
SELECT MAX(Grup.KategoriAdet)
FROM (
 SELECT
  PC.ProductCategoryID,
  COUNT(*) AS KategoriAdet
 FROM
  Production.ProductCategory PC
  INNER JOIN Production.ProductSubcategory PSC
  ON PC.ProductCategoryID = PSC.ProductCategoryID
 GROUP BY
 PC.ProductCategoryID
) Grup
```

	(No column name)
1	14

Türetilmiş tabloları daha iyi anlamak ve sorgu sonucunun sağlamasını yapmak için sorgudaki parantezler içerisindeki **SELECT** sorgusunu ayrı olarak çalıştıralım.

```
SELECT
 PC.ProductCategoryID,
 COUNT(*) AS KategoriAdet
FROM
 Production.ProductCategory PC
INNER JOIN
 Production.ProductSubcategory PSC
 ON PC.ProductCategoryID = PSC.ProductCategoryID
GROUP BY
 PC.ProductCategoryID
```

	ProductCategoryID	KategoriAdet
1	3	8
2	1	3
3	2	14
4	4	12

Bir sorgunun sonucunu türetilmiş tablo olarak kullanmak için, hesaplanmış tüm sütunlara bir takma isim verilmelidir. Eğer hesaplanmış sütunlara takma isim verilmeyecekse, tabloya verilen takma isimden sonra, tüm sütunların araları virgül ile ayrılarak yazılmalıdır.

```
SELECT MAX(Grup.KategoriAdet)
FROM (
SELECT PC.ProductCategoryID,
COUNT(*) FROM Production.ProductCategory PC
INNER JOIN Production.ProductSubcategory PSC
ON PC.ProductCategoryID = PSC.ProductCategoryID
GROUP BY PC.ProductCategoryID
) Grup (ProductCategoryID, KategoriAdet)
```

	(No column name)
1	14

Türetilmiş tabloda isimlendirmenin önemini kavramak için, **COUNT(*)** fonksiyonuna verilen takma ismi (**KategoriAdet**) ve tabloya verilen takma ismin

(**Grup**) yanında bulunan parantez içerisindeki isimleri silerek çalıştırıldığınızda hata verecektir.

```
SELECT MAX(Grup.KategoriAdet)    --Hatalı Sorgu
FROM (
 SELECT PC.ProductCategoryID,
 COUNT(*) FROM Production.ProductCategory PC
 INNER JOIN Production.ProductSubcategory PSC
 ON PC.ProductCategoryID = PSC.ProductCategoryID
 GROUP BY PC.ProductCategoryID
) Grup
```

İlişkili Alt Sorgular

İlişkili sorgular, dışarıdaki (parantez içerisindeki) sorgunun döndürdüğü her satır için, içerideki sorgunun tekrarlandığı sorgulara denir. Her satır için tekrarlanan sorgular oluşturduğu için performans açısından önerilmez.

İlişkili Alt Sorgular Nasıl Çalışır?

İlişkili alt sorguların iç içe alt sorgulardan farkı, bilgi aktarımının tek yönlü değil, çift yönlü olmasıdır. İç içe alt sorgularda iç sorgudan elde edilen bilgi, dış sorguya gönderilmektedir.

Ancak ilişkili alt sorgularda durum böyle değildir. İlişkili sorgularda, iç sorgu, dış sorgudan gelen veriyi çalıştırır ve dış sorguda da iç sorgudan elde edilen bilgiler çalıştırılır.

İlişkili iç sorguların işleyişi şu şekildedir.

- Dış sorgu, elde ettiği kayıtları iç sorguya gönderir.
- Dış sorgudan gelen değerler ile iç sorgu çalışır.
- İç sorgu, elde ettiği sonuçları tekrar dış sorguya gönderir, dış sorguda bu değerlere göre çalışarak işlemi tamamlar.

SELECT Listesindeki İlişkili Alt Sorgular

Ürünler tablosundaki alt kategori ile alt kategoriler tablosundaki alt kategorileri ilişkilendirerek, her ürünün hangi alt kategoride yer aldığını seçelim.

```
SELECT
 ProductID,
 Name,
 ListPrice,
 (SELECT
 Name
 FROM Production.ProductSubcategory AS PSC
 WHERE PSC.ProductSubcategoryID = PP.ProductSubcategoryID) AS AltKategori
FROM
 Production.Product AS PP;
```

	ProductID	Name	ListPrice	AltKategori
1	1	Adjustable Race	0,00	NULL
2	2	Bearing Ball	0,00	NULL
3	3	BB Ball Bearing	0,00	NULL
4	4	Headset Ball Bearings	0,00	NULL
5	316	Blade	0,00	NULL
6	317	LL Crankarm	0,00	NULL
7	318	ML Crankarm	0,00	NULL

WHERE Koşulundaki İlişkili Alt Sorgular

İlişkili alt sorgular, **WHERE** koşulu içerisinde de kullanılabilir.

Sistemdeki ilk gün siparişlerini ve sistemde bulunan ilk sipariş tarihini ve bu siparişlerin ID bilgilerini istiyoruz.

```
SELECT
 SOH1.CustomerID,
 SOH1.SalesOrderID,
 SOH1.OrderDate
FROM  Sales.SalesOrderHeader AS SOH1
WHERE SOH1.OrderDate = (SELECT MIN(SOH2.OrderDate)
                        FROM Sales.SalesOrderHeader AS SOH2
                        WHERE SOH2.CustomerID = SOH1.CustomerID)
ORDER BY SOH1.CustomerID;
```

	CustomerID	SalesOrderID	OrderDate
1	11000	43793	2005-07-22 00:00:00.000
2	11001	43767	2005-07-18 00:00:00.000
3	11002	43736	2005-07-10 00:00:00.000
4	11003	43701	2005-07-01 00:00:00.000
5	11004	43810	2005-07-26 00:00:00.000
6	11005	43704	2005-07-02 00:00:00.000
7	11006	43819	2005-07-27 00:00:00.000

EXISTS ve NOT EXISTS

EXISTS, SQL Server'da **SELECT** işlemi gerçekleştirmez, sadece sorgu sonucunda değer dönüp dönmediğine bakar. **EXISTS**'in çalıştığı sorguda belirtilen kriterlere uyan verinin mevcut olup olmamasına göre **TRUE** ya da **FALSE** değer elde edilir.

Daha önce kullandığımız bir sorgu üzerinde **EXIST**'i kullanalım.

Şirket içerisinde başka bir görev için başvuru yapan personellerin listesini elde edelim.

```
SELECT
 PP.BusinessEntityID,
 PP.FirstName,
 PP.LastName,
 P.PhoneNumber
FROM Person.Person PP
JOIN Person.PersonPhone AS P
ON PP.BusinessEntityID = P.BusinessEntityID
JOIN HumanResources.JobCandidate AS JC
ON P.BusinessEntityID = JC.BusinessEntityID
WHERE EXISTS(SELECT BusinessEntityID
   FROM HumanResources.JobCandidate AS JC
   WHERE JC.BusinessEntityID = PP.BusinessEntityID);
```

	BusinessEntityID	FirstName	LastName	PhoneNumber
1	212	Peng	Wu	164-555-0164
2	274	Stephen	Jiang	238-555-0197

EXISTS kullanmadığımızda da 2 kayıt listelenmişti. İki sorgu yapısı arasında, bir listeleme farkı olmadığını gördük.

EXISTS kullanılmasının sebebi; **performans**, **kod okunabilirliği** ve **sadeliktir**.

EXISTS anahtar sözcüğü, satır satır birleştirme işlemi yapmaz. İlk eşleşen değeri bulana kadar kayıtlara bakar ve bulduğunda durur. İlk eşleşme sağlandığında **TRUE** sonucunu verir. Hiç kayıt dönmezse, dışarıdaki sorgu çalıştırılmaz ve **FALSE** değeri üretir.

Veri Tiplerini Dönüştürmek: CAST ve CONVERT

SQL Server'da veri tipi dönüştürme işlemleri için kullanılan **CAST** ve **CONVERT** fonksiyonları, aynı işlevi görmektedir. **CONVERT** fonksiyonu bazı durumlarda, **CAST** fonksiyonunun gerçekleştiremediği dönüştürme işlemlerini gerçekleştirebilir.

Genel olarak aynı işleve sahip bu fonksiyonların arasındaki önemli bir fark, **CAST** fonksiyonunun **ANSI** uyumlu olmasıdır. **CONVERT** fonksiyonu ise **ANSI** uyumlu değildir.

Söz Dizimi:

```
CAST( ifade AS veri_tipi )
CONVERT( veri_tipi, ifade[, style] )
```

Dönüşüm işlemlerinde en yoğun kullanım nümerik değerlerin metinsel değere dönüştürülmesi ve tarih format dönüşümleridir.

Bir nümerik değeri metinsel değere dönüştürerek başka bir metin ile birleştirelim.

```
SELECT 'Ürün Kodu : '
   + CAST(ProductID AS VARCHAR)
   + ' - '
   + 'Ürün Adı : ' + Name
FROM Production.Product;
```

	(No column name)
1	Ürün Kodu : 1004 - Ürün Adi : % 20 indirimli ürün
2	Ürün Kodu : 1 - Ürün Adi : Adjustable Race
3	Ürün Kodu : 461 - Ürün Adi : Advanced SQL Server
4	Ürün Kodu : 879 - Ürün Adi : All-Purpose Bike Stand
5	Ürün Kodu : 712 - Ürün Adi : AWC Logo Cap
6	Ürün Kodu : 3 - Ürün Adi : BB Ball Bearing
7	Ürün Kodu : 2 - Ürün Adi : Bearing Ball

CAST ve **CONVERT** fonksiyonlarını da kullanacağımız basit bir örnek yapalım.

```
DECLARE @deger DECIMAL(5, 2);
SET @deger = 14.53;
SELECT CAST(CAST(@deger AS VARBINARY(20)) AS DECIMAL(10,5));

-- ya da CONVERT fonksiyonu ile

SELECT CONVERT(DECIMAL(10,5),
CONVERT(VARBINARY(20), @deger));
```

	(No column name)
1	14.53000

	(No column name)
1	14.53000

Common Table Expressions (CTE)

SQL Server'da geçici bir tablo oluşturma ihtiyacı durumunda, birçok farklı yöntem inceledik ve kullandık. **CTE** (*Common Table Expressions*) özelliği de, sorgu sonuçları için geçici bir tablo oluşturarak üzerinde işlem yapmaya yarayan bir özelliktir.

CTE özelliği, 2005 yılında SQL Server 2005'e eklenen bir özellik olmakla birlikte yeni sürümlerinde farklı özellikler eklenerek genişletilmiştir.

CTE'nin diğer benzeri özelliklerden farkı, öz yinelemeli (*recursive*) olmasıdır. Yazılım geliştiriciler öz yineleme kavramının ne olduğuna aşina olmalıdırlar. Özetle, bir işlemi tamamlayana kadar, kendi içerisinde aynı işlemi yineleyerek gerçekleştirmeye denir.

Söz Dizimi:

```
;WITH CTEIsmi(sutun_ismi1[, sutun_ismi2, ...]) AS
(
 Select İfadesi
)
```

Söz diziminde **WITH** deyiminin solunda bulunan noktalı virgül (;) kullanımı zorunlu değildir.

Ürün tablosu üzerinde bir CTE tanımlayalım.

```
;WITH CTEProduct(UrunNo, UrunAd, Renk) AS
(
 SELECT ProductID, Name, Color FROM Production.Product
 WHERE ProductID > 400 AND Color IS NOT NULL
)
SELECT * FROM CTEProduct;
```

	UrunNo	UrunAd	Renk
1	461	Advanced SQL Server	Silver
2	679	Rear Derailleur Cage	Silver
3	680	HL Road Frame - Black, 58	Black
4	706	HL Road Frame - Red, 58	Red
5	707	Sport-100 Helmet, Red	Red
6	708	Sport-100 Helmet, Black	Black
7	709	Mountain Bike Socks, M	White

CTE söz diziminde dikkat edilmesi gereken bir diğer konu da, **CTE** parantezlerinden sonra yazılan **SELECT** sorgusu ile parantezler arasında başka bir ifade bulunmamalıdır.

CTE ile **INSERT**, **UPDATE**, **DELETE** işlemleri de gerçekleştirilebilir.

```
WITH CTEProduct(UrunNo, UrunAd, Renk) AS
(
 SELECT ProductID, Name, Color FROM Production.Product
 WHERE ProductID > 400 AND Color IS NOT NULL
)
UPDATE CTEProduct SET UrunAd = 'Advanced SQL Server'
WHERE UrunNo = 461;
```

Güncelleme işleminin sonucunu test edelim.

```
SELECT ProductID, Name, Color
FROM Production.Product WHERE ProductID = 461;
```

	ProductID	Name	Color
1	461	Advanced SQL Server	Silver

`ProductID` değeri **`461`** olan kaydın **`UrunAd`**'ını '*Advanced SQL Server*' olarak değiştirdik. Burada dikkat ederseniz, gerçekten var olmayan tablo isimlerini CTE üzerinden kullanarak gerçek veriler ve sütunlar üzerinde gerçek bir işlem gerçekleştirdik. Ayrıca bu CTE örneğimizde **`WITH`**'den önce bir noktalı virgül (`;`) kullanmadık. Birden fazla CTE alt alta aynı sorgu içerisinde kullanılabilir.

En pahalı ürün ile en ucuz ürünü bularak bir sorgu sonucunda birleştirelim.

```
WITH EnPahaliUrunCTE
AS
(
 SELECT TOP 1 ProductID, Name, ListPrice FROM Production.Product
 WHERE ListPrice > 0
 ORDER BY ListPrice ASC
),
EnUcuzUrunCTE
AS
(
 SELECT TOP 1 ProductID, Name, ListPrice FROM Production.Product
 ORDER BY ListPrice DESC
)
SELECT * FROM EnPahaliUrunCTE
 UNION
SELECT * FROM EnUcuzUrunCTE;
```

	ProductID	Name	ListPrice
1	749	Road-150 Red, 62	4105,5685
2	873	Patch Kit/8 Patches	2,6275

Rütbeleme Fonksiyonları ile Kayıtları Sıralamak

SQL Server'da bir kayıt listesi oluşturduğunuzda, bu liste içerisinde birçok farklı sebep ile çeşitli sıralamalar yapmak isteyebilirsiniz. Bu sıralama işlemi, gruplara ayrılmış verilerin farklı sayılar ile belirtilmesi olabileceği gibi, tüm satırlar için bir numaralandırma da olabilir.

Bu bölüm, bu tür sıralama işlemlerini içermektedir.

ROW_NUMBER()

Belirtilen ifadeye göre satırları sıralar ve her bir satır için artan numaraların bulunduğu bir sütun üretir.

Ürünleri listeleyelim ve her bir kayıt için bir sıra numarası oluşturarak ilk sütun olarak sıralayalım.

```
SELECT ROW_NUMBER() OVER(ORDER BY ProductID) AS SatirNO,
 ProductID, Name, ListPrice
FROM Production.Product;
```

	SatirNO	ProductID	Name	ListPrice
1	1	1	Adjustable Race	0,00
2	2	2	Bearing Ball	0,00
3	3	3	BB Ball Bearing	0,00
4	4	4	Headset Ball Bearings	0,00
5	5	316	Blade	0,00
6	6	317	LL Crankarm	0,00
7	7	318	ML Crankarm	0,00

1
2
3
4
5
6
7

SSMS editörünün sonuç ekranında kayıtların sırasını gösteren tablonun en sağında bulunan ve artan şekilde sıralanan sütun da bu fonksiyona bir örnektir. SSMS de arka planda bu tür bir sorgu kullanarak bu işlemi gerçekleştirir.

RANK ve DENSE_RANK Fonksiyonları

RANK ve **DENSE_RANK** fonksiyonları kayıtları listelerken her bir kayıt için bir sıra numarası vermeye yarar.

RANK

RANK fonksiyonu, bir veri kümesi içinde gruplama yaparak her bir veriyi belirtilen kritere göre sıralama yaparak numaralandırır.

Ancak **ROW_NUMBER()** fonksiyonundan küçük ve belirgin bir farkı vardır. **RANK** ile sıralanan kayıtlar arasında, aynı değerlere sahip kayıtlar var ise, bu kayıtlara aynı sıra numaralarını verir.

```
SELECT Inv.ProductID, P.Name,
    Inv.LocationID, Inv.Quantity,
    RANK() OVER(PARTITION BY Inv.LocationID
      ORDER BY Inv.Quantity DESC) AS 'RANK'
    FROM Production.ProductInventory Inv
    INNER JOIN Production.Product P
    ON Inv.ProductID = P.ProductID;
```

Örnekte görüldüğü gibi, aynı Quantity değerlerine sahip kayıtlarda sıra numarası da aynı verilmiştir.

	ProductID	Name	LocationID	Quantity	RANK
1	389	Hex Nut 2	1	657	1
2	367	Thin-Jam Hex Nut 3	1	643	2
3	413	Internal Lock Washer 4	1	641	3
4	440	Lock Nut 16	1	640	4
5	439	Lock Nut 6	1	636	5
6	396	Hex Nut 18	1	636	5
7	441	Lock Nut 17	1	635	7

DENSE_RANK

RANK fonksiyonu ile benzer işleve sahiptir. İki fonksiyon arasındaki fark; **RANK** fonksiyonu benzer değerlere sahip kayıtlara aynı sıra numarası ile sıralandırırken, **DENSE_RANK** fonksiyonu aynı kayıtlar da bile, farklı sıra numarası vererek sıralar.

RANK fonksiyonu sonucunu tekrar incelerseniz, aynı değerlere sahip kayıtların sıra numarasının da aynı olduğunu görebilirsiniz.

RANK fonksiyonu için yaptığımız örneği **DENSE_RANK** için tekrar yapalım.

```
SELECT Inv.ProductID, P.Name,
  Inv.LocationID, Inv.Quantity,
       DENSE_RANK() OVER(PARTITION BY Inv.LocationID
  ORDER BY Inv.Quantity DESC) AS 'DENS_RANK'
  FROM Production.ProductInventory Inv
  INNER JOIN Production.Product P
  ON Inv.ProductID = P.ProductID;
```

	ProductID	Name	LocationID	Quantity	DENS_RANK
1	389	Hex Nut 2	1	657	1
2	367	Thin-Jam Hex Nut 3	1	643	2
3	413	Internal Lock Washer 4	1	641	3
4	440	Lock Nut 16	1	640	4
5	439	Lock Nut 6	1	636	5
6	396	Hex Nut 18	1	636	5
7	441	Lock Nut 17	1	635	6

NTILE

NTILE fonksiyonu diğer fonksiyonlara göre biraz daha ileri seviye ve fonksiyoneldir. Dışarıdan aldığı parametreye göre her bir grup içerisindeki veriyi hesaplar.

Personel bilgilerinin bulunduğu tablo üzerinde bir birleştirme sorgusu gerçekleştirerek elde edilecek sonucu inceleyelim.

```
SELECT P.FirstName, P.LastName,
  S.SalesYTD, A.PostalCode,
  NTILE(4) OVER(ORDER BY SalesYTD DESC) AS 'NTILE'
FROM Sales.SalesPerson S
INNER JOIN Person.Person P ON S.BusinessEntityID =
P.BusinessEntityID
INNER JOIN Person.Address A ON A.AddressID = P.BusinessEntityID;
```

	FirstName	LastName	SalesYTD	PostalCode	NTILE
1	Linda	Mitchell	4251368,5497	98027	1
2	Jae	Pak	4116871,2277	98055	1
3	Michael	Blythe	3763178,1787	98027	1
4	Jillian	Carson	3189418,3662	98027	1
5	Ranjit	Varkey Chudukatil	3121616,3202	98055	1
6	José	Saraiva	2604540,7172	98055	2
7	Shu	Ito	2458535,6169	98055	2

TABLESAMPLE

SQL Server'da çoğu zaman, **RANDOM** veri üzerinde çalışma ihtiyacı oluşur. Rastgele gelen sayı ve sıradaki veri kullanılır.

Bu işlemi gerçekleştirmek için genel olarak **NEWID()** fonksiyonu kullanılır.

```
SELECT TOP 20 Name,
  ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY NEWID();
```

	Name	ProductNumber	ReorderPoint
1	ML Touring Seat Assembly	SA-T612	375
2	Rear Brakes	RB-9231	375
3	Hex Nut 3	HN-6320	750
4	Taillights - Battery-Powered	LT-T990	3
5	External Lock Washer 1	LE-6000	750
6	Hex Nut 2	HN-5400	750
7	Mountain-400-W Silver, 38	BK-M38S-38	75

Ancak oluşan yeni ihtiyaçlar neticesinde **TABLESAMPLE** adına verilen bir özellik geliştirildi. Bu özellik ile istenen tablodaki verinin yüzde (%) olarak belirli bir kısmı, ya da kayıtları sayı ile listeleme gerçekleştirilebilir.

TABLESPACE'in yüzde ifadesi ile kullanımı;

Ürünler tablosundaki kayıtların yarısını (%50) listeleyelim.

```
SELECT * FROM Production.Product TABLESAMPLE(50 PERCENT);
```

	ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	368	Thin-Jam Hex Nut 4	HJ-5162	0	0	NULL	1000	750	0,00	0,00
2	369	Thin-Jam Hex Nut 13	HJ-5811	0	0	NULL	1000	750	0,00	0,00
3	370	Thin-Jam Hex Nut 14	HJ-5818	0	0	NULL	1000	750	0,00	0,00
4	371	Thin-Jam Hex Nut 7	HJ-7161	0	0	NULL	1000	750	0,00	0,00
5	372	Thin-Jam Hex Nut 8	HJ-7162	0	0	NULL	1000	750	0,00	0,00
6	373	Thin-Jam Hex Nut 12	HJ-9080	0	0	NULL	1000	750	0,00	0,00
7	374	Thin-Jam Hex Nut 11	HJ-9161	0	0	NULL	1000	750	0,00	0,00

Sorguyu her yenilediğinizde fark edeceğiniz önemli bir durum şudur. Her zaman farklı sayıda ve içerikteki veri listelenecektir. Bu durumda aslında gerçek anlamda kayıtların yarısını göremezsiniz. Bunu SQL Server kendisi belirler. Siz sadece, bir oran belirtmiş olursunuz.

Yüzde ifadesi ile yapılan bu işlem, kayıt sayısı olarak da gerçekleştirilebilir.

```
SELECT * FROM Production.Product TABLESAMPLE(300 ROWS);
```

	ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	1	Adjustable Race	AR-5381	0	0	NULL	1000	750	0,00	0,00
2	2	Bearing Ball	BA-8327	0	0	NULL	1000	750	0,00	0,00
3	3	BB Ball Bearing	BE-2349	1	0	NULL	800	600	0,00	0,00
4	4	Headset Ball Bearings	BE-2908	0	0	NULL	800	600	0,00	0,00
5	316	Blade	BL-2036	1	0	NULL	800	600	0,00	0,00
6	317	LL Crankarm	CA-5965	0	0	Black	500	375	0,00	0,00
7	318	ML Crankarm	CA-6738	0	0	Black	500	375	0,00	0,00

Ürünler tablosunda 505 kayıt bulunmaktadır. 300 kayıtlık bir istekte bulunulduğunda yüzde ifadesinde olduğu gibi net bir geri dönüş sayısı ve içerikle karşılaşmayız. Bu sorgu sonucunda geri dönen sayısı her sorguda değişecektir.

Tabloda 505 kayıt var ve sorguda 600 kayıt listelemek isteseydik ne olurdu?

```
SELECT * FROM Production.Product TABLESAMPLE(600 ROWS)
```

	ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	1	Adjustable Race	AR-5381	0	0	NULL	1000	750	0,00	0,00
2	2	Bearing Ball	BA-8327	0	0	NULL	1000	750	0,00	0,00
3	3	BB Ball Bearing	BE-2349	1	0	NULL	800	600	0,00	0,00
4	4	Headset Ball Bearings	BE-2908	0	0	NULL	800	600	0,00	0,00
5	316	Blade	BL-2036	1	0	NULL	800	600	0,00	0,00
6	317	LL Crankarm	CA-5965	0	0	Black	500	375	0,00	0,00
7	318	ML Crankarm	CA-6738	0	0	Black	500	375	0,00	0,00

Tüm kayıtların tamamı 505 olması ve bu kayıtlardan daha fazlasına ihtiyacımız olduğunu belirtmemiz nedeniyle, SQL Server bize tüm kayıtları listeleyecektir. Yani, bu sorgu ile 505 kaydın tamamı listelenir.

Sürekli değişen içerik ve sayıda veri ile çalışmak istemeyebilirsiniz. Hatta verdiğiniz değere göre çalışacak bu sorgu, bazen sık sık boş kayıt listesi dönebilir. Uygulamanızda bu tür sorunları yaşamamak için, veritabanından alınan sorgu sonuçlarının sabit kalmasını sağlayabilirsiniz.

```
SELECT FirstName, LastName
FROM Person.Person TABLESAMPLE(300 ROWS)
REPEATABLE(300);
```

	FirstName	LastName
1	Alisha	Lin
2	Alvin	Lin
3	Amy	Lin
4	Arturo	Lin
5	Autumn	Lin
6	Barbara	Lin
7	Bianca	Lin

Yukarıdaki sorgu ile veritabanından 300 satırlık bir kayıt isteniyor. Tabi ki, **TABLESPACE** yapısı gereği farklı bir değer üretecektir. Ancak, **REPEATABLE()** kullanarak bu en son alınan sorgu sonucunun sabitlenmesi sağlanabilir.

TABLESPACE ve **REPEATABLE** ile birlikte **ROWS** kullanıldığı gibi **PERCENT**'de kullanılabilir.

PIVOT ve UNPIVOT Operatörleri

PIVOT ve **UNPIVOT** operatörleri, dışarıdan bir tablo değeri girdi olarak alır ve satırları sütunlara ya da sütunları satırlara dönüştürerek yeni bir tablo değeri oluştururlar. **FROM** yan cümleciği ile birlikte kullanılırlar.

PIVOT

PIVOT operatörü, önemli bir özellik olmakla birlikte, en çok kullanıldığı alanlar **OLAP** türü sorgular ve açık şema uygulamalarıdır.

Açık şema uygulamalar, ileri seviye ve karmaşık yapıya sahip veritabanı uygulamalarına denir. Birçok ürün satan ve her ürünün alt bir çok özelliği bulunan veritabanlarında (e-ticaret, bankacılık vb.), ürün ya da benzeri nesnelerin alt özellikleri farklı bir tabloda satırlar halinde tutulur. İhtiyaç halinde **PIVOT** ile sütunlara dönüştürülürler.

Bildiğiniz gibi **AdventureWorks** veritabanı bir bisiklet satışı yapan firmanın veritabanı tasarım modelidir. Şimdi kayıtlı bisikletlerimiz için şöyle bir örnek yapalım.

Tüm bisikletlerden hangi renkte, kaç adet olduğunu listeleyelim.

```
SELECT * FROM
(
 SELECT PSC.Name, P.Color, Envanter.Quantity
 FROM Production.Product P
 INNER JOIN Production.ProductSubcategory PSC
 ON PSC.ProductSubcategoryID = P.ProductSubcategoryID
 LEFT JOIN Production.ProductInventory Envanter
 ON P.ProductID = Envanter.ProductID
)Tablom
PIVOT
(
 SUM(Quantity)
 FOR Color
 IN([Black],[Red],[Blue],[Multi],[Silver],[Grey],[White],[Yellow],
[Silver/Black])
)PivotTablom;
```

	Name	Black	Red	Blue	Multi	Silver	Grey	White	Yellow	Silver/Black
1	Bib-Shorts	NULL	NULL	NULL	324	NULL	NULL	NULL	NULL	NULL
2	Bike Racks	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	Bike Stands	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	Bottles and Cages	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	Bottom Brackets	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
6	Brakes	NULL	NULL	NULL	NULL	1490	NULL	NULL	NULL	NULL
7	Caps	NULL	NULL	NULL	288	NULL	NULL	NULL	NULL	NULL

UNPIVOT

PIVOT işleminin tam tersi işleve sahiptir. **PIVOT** satırları sütunlara dönüştürürken **UNPIVOT** ise, sütunları satırlara dönüştürmek için kullanılır.

UNPIVOT işlemini anlatan bir örnek yapalım.

UnPvt isminde bir tablo oluşturalım. Bu tablonun sütunlarını satır haline getireceğiz.

```
CREATE TABLE UnPvt(
 VendorID int, Col1 int, Col2 int,
    Col3 int, Col4 int, Col5 int);
```

Oluşturulan tabloya kayıt ekleyelim.

```
INSERT INTO UnPvt VALUES (1,4,3,5,4,4);
INSERT INTO UnPvt VALUES (2,4,1,5,5,5);
INSERT INTO UnPvt VALUES (3,4,3,5,4,4);
INSERT INTO UnPvt VALUES (4,4,2,5,5,4);
INSERT INTO UnPvt VALUES (5,5,1,5,5,5);
```

Oluşturarak içerisine veri eklediğimiz tabloyu listeleyerek sütunlarını inceleyelim.

```
SELECT * FROM UnPvt;
```

	VendorID	Col1	Col2	Col3	Col4	Col5
1	1	4	3	5	4	4
2	2	4	1	5	5	5
3	3	4	3	5	4	4
4	4	4	2	5	5	4
5	5	5	1	5	5	5

Eklenen kayıtlar ile birlikte tabloya `UNPIVOT` işlemi uygulayalım.

```
SELECT VendorID, Employee, Orders
FROM
   (SELECT VendorID, Col1, Col2, Col3, Col4, Col5 FROM UnPvt) p
 UNPIVOT
   (Orders FOR Employee IN (Col1, Col2, Col3, Col4, Col5)
)AS unpvt_table;
```

	VendorID	Col1	Col2	Col3	Col4	Col5
1	1	4	3	5	4	4
2	2	4	1	5	5	5
3	3	4	3	5	4	4
4	4	4	2	5	5	4
5	5	5	1	5	5	5

UNPIVOT işleminden önce ve sonraki görünümü inceleyerek, **UNPIVOT** işleminin sütunları satırlara dönüştürme işlemini nasıl yaptığını daha iyi kavrayabilirsiniz.

INTERSECT

İki farklı sorgu sonucunun kesişimini elde etmek için kullanılır. Yani, iki sorgu sonuç kümesinde de ortak olan verilerin gösterilmesi için kullanılır.

```
SELECT ProductCategoryID
FROM Production.ProductCategory
INTERSECT
SELECT ProductCategoryID
FROM Production.ProductSubcategory
```

	ProductCategoryID
1	1
2	2
3	3
4	4

Production.ProductCategory tablosunda 4 adet ana kategori bulunmaktadır. Bu kategoriler ile ilişkili alt kategorilerin tutulduğu **Production.ProductSubCategory** tablosundaki **ProductCategoryID** sütununu **INTERSECT** operatörü ile kesişim işlemine tabi tuttuğumuzda, iki tabloda kesişen toplam 4 kategori olduğunu görüyoruz.

SQL Server'da hemen her işlem için birden fazla çözüm yolu bulunmaktadır. **INTERSECT** operatörünün gerçekleştirdiği işlemi de **IN** ya da **EXISTS** operatörünü kullanarak da gerçekleştirebiliriz.

IN ile;

```
SELECT ProductCategoryID
FROM Production.ProductCategory
WHERE ProductCategoryID IN(
  SELECT ProductCategoryID
  FROM Production.ProductSubCategory);
```

Benzer şekilde **EXISTS** ile de gerçekleştirilebilir.

EXCEPT

EXCEPT operatörü, iki farklı sorgu sonucunu karşılaştırır ve sadece ilk sonuç setinde olan, ikinci sorgu sonucunda olmayan kayıtları listelenmesini sağlar.

```
SELECT ProductID
FROM Production.Product
EXCEPT
SELECT ProductID
FROM Production.WorkOrder;
```

	ProductID
1	863
2	862
3	861
4	860
5	859
6	858
7	878

TRUNCATE TABLE İLE VERİ SİLMEK

Bir tablodaki verilerin tamamını silmek için kullanılır. Amaç olarak **DELETE** komutu ile aynı olsa da çalışma olarak farklıdır. Bir **DELETE** işleminde koşul belirterek bir ya da belirlenen kayıtların silinmesi gerçekleştirilebilirken **TRUNCATE TABLE** kullanılan bir silme işleminde koşul belirtilmez. Bu işlem tablodaki tüm kayıtların silinmesini sağlar.

DELETE komutu, silinmek istenen veriyi koşullara göre silme özelliğine sahip olduğu için, tüm satırları tek tek siler. Ancak **TRUNCATE TABLE** komutu, tüm kayıtları veritabanından bağı kopartılarak sildiği için hızlı çalışır. **DELETE** komutu ile silinen kayıtlar, veritabanı mimarisi geri alma işlemini gerçekleştirecek şekilde tasarlandı ise, silinen kayıtlar geri alınabilir. Ancak **TRUNCATE TABLE** kullanılarak silinen kayıtlar geri alınamaz.

Söz Dizimi:

```
TRUNCATE TABLE table_name
```

Production.Product tablosundaki tüm verileri silmek için;

```
TRUNCATE TABLE Production.Product;
```

İleri Veri Yönetim Teknikleri

Veri Ekleme

Veritabanına bir kayıt eklerken **INSERT** ifadesi kullanılır. Bu işlem temel anlamda basit olsa da, ileri seviye veri ekleme ve yönetim özellikleri mevcuttur. Bu bölümde, veri eklemek için esnekleştirilen ve geliştirilen özellikleri öğreneceksiniz.

Sorgu Sonucunu Yeni Tabloda Saklamak

Bir seçme ifadesi ile alınan sonucu, önceden var olmayan yeni bir tabloda saklamak için, **SELECT** ifadesi ile birlikte **INTO** deyimi kullanılır.

Bu şekilde gerçek değil, sanal bir geçici tablo oluşturulur. Hızlı geçici tablo oluşturmak ve kullanmak için etkili bir tekniktir.

Söz Dizimi:

```
SELECT sutun_isimleri
INTO #gecici_tablo_ismi
FROM tablo_ismi
```

Person.Person tablosundan aldığımız veriyi, geçici bir tablo oluşturarak aktaralım.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName
INTO #personeller
FROM Person.Person;
```

```
(19972 row(s) affected)
```

Şimdi de geçici tabloyu sorgulayarak kayıtları listeleyelim.

```
SELECT * FROM #Personeller;
```

	BusinessEntityID	FirstName	MiddleName	LastName
1	285	Syed	E	Abbas
2	293	Catherine	R.	Abel
3	295	Kim	NULL	Abercrombie
4	2170	Kim	NULL	Abercrombie
5	38	Kim	B	Abercrombie
6	211	Hazem	E	Abolrous
7	2357	Sam	NULL	Abolrous

Stored Procedure Sonucunu Tabloya Eklemek

SQL Server'da veri ekleme ve üzerinde düzenleme işlemleri çok önemli ve çok kullanılan işlemlerdir. Bu nedenle, veri ekleme işlemlerinde, bazı ek özellikler ile esneklik sağlanmaktadır. Stored Procedure'den dönen sonucu bir tabloya eklemek (**INSERT**)'de bu esnekliklerden biridir.

Oluşturduğunuz bir Stored Procedure'ün sonuçlarını, veri ekleyeceğiniz tablonun yapısına uygun olacak şekilde bir tabloda saklayabilirsiniz.

Söz Dizimi:

```
INSERT INTO tablo_yada_view [(sutun_listesi)]
EXEC sp_adi
```

Personellerin bulunduğu **Person.Person** tablosundaki kayıtların tamamını yeni oluşturacağımız **Personeller** adındaki bir tabloya aktaralım. Ancak bu aktarımı yaparken tüm sütunları almak yerine, sadece önemli ve ihtiyacımız olan sütunları aktaralım.

Verileri aktaracağımız tabloyu oluşturalım.

```
CREATE TABLE Personeller
(
 BusinessEntityID INT,
 FirstName  VARCHAR(50),
 MiddleName  VARCHAR(50),
 LastName  VARCHAR(50)
);
```

Oluşturduğumuz tabloya **Person.Person** tablosundan veri çekeceğiz. Bu işlemi gerçekleştirecek Stored Procedure'ü geliştirelim.

```
CREATE PROC pr_GetAllPerson
AS
BEGIN
 SELECT BusinessEntityID, FirstName, MiddleName, LastName
 FROM Person.Person;
END;
```

Prosedürümüzü test edelim.

```
EXEC pr_GetAllPerson;
```

	BusinessEntityID	FirstName	MiddleName	LastName
1	285	Syed	E	Abbas
2	293	Catherine	R.	Abel
3	295	Kim	NULL	Abercrombie
4	2170	Kim	NULL	Abercrombie
5	38	Kim	B	Abercrombie
6	211	Hazem	E	Abolrous
7	2357	Sam	NULL	Abolrous

Şimdi de prosedürümüzden gelen veriyi **Personeller** tablosuna aktaralım.

```
INSERT INTO Personeller(BusinessEntityID, FirstName, MiddleName,
LastName)
EXEC pr_GetAllPerson;
```

Son olarak, **Personeller** tablomuzdaki kayıtları listeleyelim.

```
SELECT * FROM Personeller;
```

	BusinessEntityID	FirstName	MiddleName	LastName
1	285	Syed	E	Abbas
2	293	Catherine	R.	Abel
3	295	Kim	NULL	Abercrombie
4	2170	Kim	NULL	Abercrombie
5	38	Kim	B	Abercrombie
6	211	Hazem	E	Abolrous
7	2357	Sam	NULL	Abolrous

Sorgu Sonucunu Var Olan Tabloya Eklemek

Veri eklemek için sadece dışarıdan parametre almak ya da prosedürden dönen sonuçları kaydetmek zorunda değiliz. Prosedür kullanmadan bir **SELECT** sorgusunun sonucunu da başka bir tabloya aktararak, veri kaydı yapabiliriz. **INSERT** ile veri ekleme konusunda temel olarak işlediğimiz bu konu için bir örnek yapalım.

Prosedür ile yaptığımız **Person.Person** tablosundan **Personeller** tablosuna kayıt aktarma işlemini basit bir **SELECT** ile gerçekleştirelim.

```
INSERT INTO Personeller(BusinessEntityID, FirstName, MiddleName,
LastName)
  SELECT BusinessEntityID, FirstName,
    MiddleName, LastName
  FROM Person.Person;
```

SELECT sorgusu ile, **Personeller** tablosu tekrar listelendiğinde, prosedür ile gerçekleştirilen işlemin **SELECT** ile de başarılı bir şekilde gerçekleştiği görülebilir.

Veri Güncelleme

Veri güncelleme işleminde daha karmaşık işlemleri gerçekleştirebilmek için, tablo birleştirme ve alt sorgular ile veri güncelleme işlemi gerçekleştirilebilir.

Tabloları Birleştirerek Veri Güncellemek

Güncelleme işleminin tek başına yeterli olmadığı durumlar olabilir. Birden fazla tabloyu birleştirmek, bu tablolara ve sonuçlarına göre güncelleme işlemi yapmak gibi durumlar söz konusu olabilir.

Birden fazla tablonun birleştirilerek güncelleme işlemi yapılması mümkündür. Ancak, biraz karmaşık bir yapıdadır.

Söz Dizimi:

```
UPDATE tablo_ismi
SET sutun_ismi = deger | ifade
FROM tablo_ismi JOIN tablo_ismi2
ON birlestirme_ifadesi
WHERE kosul[lar]
```

Veritabanımızda daha önce satın alınmış ürünler **Sales.SalesOrderDetail** tablosunda yer alır. Bu tabloyu kullanarak, en az 1 kez satın alınan, yani bu tabloda yer alan ürünlerde **%4** zaman yapalım.

```
UPDATE Production.Product
SET ListPrice = ListPrice * 1.04
FROM Production.Product PP JOIN Sales.SalesOrderDetail SOD
ON PP.ProductID = SOD.ProductID;
```

Bir **UPDATE** ifadesi aynı anda sadece bir tabloya ait sütunları güncelleyebilir.

Alt Sorgular İle Veri Güncellemek

Bir güncelleme işleminde alt sorgular da kullanılabilir.

Önceki örnekte yaptığımız **%4**'lük zammı alt sorgu kullanarak yapalım.

```
UPDATE Production.Product
SET ListPrice = ListPrice * 1.04
FROM (Production.Product PP JOIN Sales.SalesOrderDetail SOD
      ON PP.ProductID = SOD.ProductID);
```

Büyük Boyutlu Verileri Güncellemek

SQL Server'da büyük boyutlu verileri de güncelleyerek yeni değerler atayabiliriz.

Söz Dizimi:

```
UPDATE tablo_yada_view_ismi
SET sutun_ismi.WRITE(ifadeler, @OffSet, @Length)
FROM tablo_ismi
WHERE kosullar
```

Söz dizimi yapısal olarak değişiklik gösterecektir. Ancak temel özellikler ile kullanımı bu şekildedir.

Büyük boyutlu veriler için kullanılabilecek veri tipleri;

- **VARCHAR(MAX) :** Değişken uzunlukta, Non-Unicode veriler için.
- **NVARCHAR(MAX) :** Değişken uzunlukta, Unicode veriler için.
- **VARBINARY(MAX) :** Değişken uzunlukta, Binary veriler için.

Örnek için kullanacağımız ve website domain ile açıklama bilgilerini tutacak bir tablo oluşturalım.

```
CREATE TABLE WebSites
(
 SiteID INT NOT NULL,
 URI  VARCHAR(40),
 Description VARCHAR(MAX)
);
```

`WebSites` tablosuna yeni bir kayıt ekleyelim.

```
INSERT INTO WebSites(SiteID, URI, Description)
VALUES(1,'www.dijibil.com','Online eğitim');
```

Eklenen kaydı görüntüleyelim.

```
SELECT * FROM WebSites WHERE SiteID = 1;
```

	SiteID	URI	Description
1	1	www.dijibil.com	Online egitim

Eklediğimiz kayıtta bir eksik tespit ettik. '*Online eğitim*' yazısının sonuna '*sistemi*' yazısını da eklemek istiyoruz.

```
UPDATE WebSites
SET Description.WRITE(' sistemi',NULL,NULL)
WHERE SiteID = 1
```

Güncellenen kaydı görüntüleyelim.

```
SELECT * FROM WebSites WHERE SiteID = 1;
```

	SiteID	URI	Description
1	1	www.dijibil.com	Online egitim sistemi

Açıklama kısmında yazdığımız '*Online*' kelimesini Türkçeleştirerek '*Çevrimiçi*' yazacağız.

```
UPDATE WebSites
SET Description.WRITE('Çevrimiçi',0,6)
WHERE SiteID = 1;
```

Son güncelleme işleminden sonra kaydımızın son halini listeleyelim.

```
SELECT * FROM WebSites WHERE SiteID = 1;
```

	SiteID	URI	Description
1	1	www.dijibil.com	Çevrimiçi egitim sistemi

Tek seferde hem normal bir sütun, hem de büyük veri içeren bir sütunu güncellemek için;

```
UPDATE WebSites
SET Description.WRITE('Online',0,9),
 URI = 'http://www.dijibil.com'
WHERE SiteID = 1;
```

Son güncelleme işleminden sonra kaydımızın son halini listeleyelim.

```
SELECT * FROM WebSites WHERE SiteID = 1;
```

	SiteID	URI	Description
1	1	http://www.dijibil.com	Online egitim sistemi

Veri Silme

Veritabanında veri silmek için **DELETE** komutu kullanılır. Genel olarak **DELETE** sorguları temel ve basit şekilde hazırlanır. Çünkü genel olarak bir veri silme işleminde çok karmaşık işlemler kullanılmaz.

Bu bölümde, birden fazla tablonun birleştirilmesiyle belirlenecek kayıtların silinmesi işlemlerine değineceğiz.

Tablo Birleştirerek Veri Silmek

Birden fazla tabloyu birleştirerek, eşleşen sonuçların silindiği bir **DELETE** sorgusu oluşturalım.

Bunun için kendi tablolarımızı oluşturarak veriler gireceğiz.

Personel bilgilerinin tutulduğu tabloyu oluşturalım.

```
CREATE TABLE Personeller
(
 PersonID INT NOT NULL,
 FirstName VARCHAR(30),
 LastName VARCHAR(30),
 JobID  INT
);
```

Personellerin görevlerinin bulunduğu tabloyu oluşturalım.

```
CREATE TABLE Jobs
(
 JobID INT NOT NULL,
 JobName VARCHAR(50)
);
```

Hazırladığımız tablolara veri girmeye başlayabiliriz. İlk olarak **Jobs** tablosuna birkaç görev adı ekleyelim.

```
INSERT INTO Jobs(JobID, JobName)
VALUES(1,'DBA'),(2,'Software Developer'),(3,'Interface Designer');
```

Görevlerimiz hazır. Şimdi yeni bir personeller ekleyelim.

```
INSERT INTO Personeller(PersonID, FirstName, LastName, JobID)
VALUES(1,'Cihan','Özhan',1),(2,'Kerim','Fırat',2),(3,'Uğur','Gelişken',2);
```

Tablolardaki verileri listeleyelim.

```
SELECT * FROM Jobs;
```

	JobID	JobName
1	1	DBA
2	2	Software Developer
3	3	Interface Designer

```
SELECT * FROM Personeller;
```

	PersonID	FirstName	LastName	JobID
1	1	Cihan	Özhan	1
2	2	Kerim	Firat	2
3	3	Ugur	Gelisken	2

Şimdi **JOIN** ile ilişkili bir sorgu hazırlayıp **DELETE** ile sileceğimiz kaydı bulalım.

```
SELECT
 P.PersonID, P.FirstName,
 P.LastName , J.JobName
FROM
 Personeller AS P
JOIN Jobs AS J
ON P.JobID = J.JobID
WHERE P.JobID = 2;
```

	PersonID	FirstName	LastName	JobName
1	2	Kerim	Firat	Software Developer
2	3	Ugur	Gelisken	Software Developer

Bu sorgu ile **JobID** değeri **1** olan kayıt ya da kayıtları ilişkili bir şekilde sorgulayarak listeledik.

Son olarak, bu kaydı **DELETE** komutunu kullanarak silelim.

```
DELETE FROM Personeller
FROM Personeller AS P
JOIN Jobs AS J
ON P.JobID = J.JobID
WHERE P.JobID = 2;
```

Bu sorgu sonucunda, **JobID** değeri 2 olan kayıtlar silinecektir.

Ancak bu sorguyu şu şekilde değiştirirsek, **PersonID** değeri 1 olan kaydı sileriz. **PersonID**, gerçek iş uygulamalarında **IDENTITY** olacağı için, tek bir kayıt silinecektir. Yani, silinecek kayıtları belirlemek için sorgu kapsamını genişletmek ya da daraltmak tamamen sizin elinizdedir.

```
DELETE FROM Personeller
FROM Personeller AS P
JOIN Jobs AS J
ON P.JobID = J.JobID
WHERE P.PersonID = 1;
```

Alt Sorgular İle Veri Silmek

Bir alt sorguya bağlı olarak kayıt silme işlemleri gerçekleştirilebilir.

Alt sorgu ile silme işlemlerine örnek olarak;

```
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM Sales.SalesPerson
     WHERE SalesYTD > 2500000.00);
```

TOP Fonksiyonu İle Veri Silmek

TOP fonksiyonu kullanarak, bir tablo üzerinde yüzde ile oransal ya da sayı ile ifade ederek belli oranda kayıt silinebilir.

Yüzde (**%**) ile kayıt silmek için;

```
DELETE TOP(2.2) PERCENT FROM Production.ProductInventory;
```

Bu işlem sonucunda **Production.ProductInventory** tablonuzdaki kayıt sayısına göre değişmekle birlikte, tablo üzerinde daha önce silme işlemi gerçekleştirmediyseniz, 27 kayıt silinecektir.

Sayı ile kayıt silmek için;

```
DELETE TOP(2) FROM Production.ProductInventory;
```

Bu sorguyu çalıştırdığınızda **Production.ProductInventory** tablosundan 2 kayıt silinecektir.

Silinen Bir Kaydın DELETED İçerisinde Görüntülenmesi

Silinen bir kaydı, silindi anda, ekranda görüntülemek için;

```
DELETE Sales.ShoppingCartItem
OUTPUT DELETED.*
WHERE ShoppingCartID = 14951;
```

Dosyaların Veritabanına Eklenmesi ve Güncellenmesi

SQL Server 2005 ve sonrasında mimari olarak gelişen SQL Server, veri depolama ve yönetimi anlamında da birçok farklı yeteneğe kavuştu. Veri denilince akla ilk gelen yapısal veriler, artık veri kelimesi için yetersiz gelmeye başladı. Bir video, müzik, resim, doküman dosyası da veri olarak kabul edilebilmeliydi. Birçok büyük veritabanı yönetim sistemi de, bu tür dosyaları veritabanında tutabilmek için bazı özellikler geliştirdi. SQL Server'da bu güce sahip olmak için **OPENROWSET** ve **FILESTREAM** özelliklerini geliştirdi.

Bu bölümde, **OPENROWSET** ve **FILESTREAM** özelliklerini kullanarak, bir resim, doküman ya da video dosyasını veritabanında depolamayı ve bu verileri yönetmeyi inceleyeceğiz.

OPENROWSET Komutu

Microsoft, SQL Server 2005 versiyonu ile birlikte birçok yeni özelliğe sahip hale geldi. Büyük veritabanlarının olması gereken ve en çok ihtiyaç duyulan özelliklerinden birisi kuşkusuz resim, video gibi '*Big Data*' denilen dosya saklama özelliğidir.

SQL Server'a 2005'ten itibaren kazandırılan özelliklerden birisi de bu büyük veri dosyalarının saklanabilmesiydi. Bu işlemi gerçekleştirmek için **OPENROWSET** komutu geliştirildi. Bu bölümde, büyük verilerin SQL Server içerisinde nasıl kullanıldığını ve yönetildiğini göreceğiz.

Söz Dizimi:

```
OPENROWSET( BULK 'data_file', SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB )
```

- **Data_File:** Veritabanına kaydedilecek dosyanın yolu.
- **SINGLE_BLOB:** Verinin Binary olarak okunacağını belirtir. (*BLOB = Binary Large Object*)
- **SINGLE_CLOB:** Karakter tipli okuma işlemi için belirtilir. (*CLOB = Character Large Object*)
- **SINGLE_NCLOB:** Karakter tipli okuma işlemi için belirtilir. (*NCLOB = National Character Large Object*)

OPENROWSET komutunun kullanımını en temel haliyle görmek istersek;

```
SELECT BulkColumn
FROM OPENROWSET(BULK 'C:\video.mp4', SINGLE_BLOB) AS Files;
```

	BulkColumn
1	0x00000018667479706D70343200000000069736F6D6D70343200114926D6F6F760000006C6D766864000000000CC327FE1CC327FE1000002580001FCB60...

Bu sorgu çalıştırıldığında, veritabanında bulunmayan, **C:** dizinindeki **video.mp4** dosyasının Binary formatındaki değerine ulaşılır. Bu dönüştürme işlemini gerçekleştiren, tabi ki SQL Server'dır.

Şimdi, gerçek bir veritabanı örneği oluşturarak **OPENROWSET** komutunun kullanımını inceleyelim.

Kullanıcıların dosyalarını(video, resim vb.) tutan bir tablo oluşturalım.

```
CREATE TABLE UserFiles
(
 UserID INT NOT NULL,
 UserFile VARBINARY(MAX) NOT NULL
);
```

UserID değeri 1 olan kullanıcımızın, veritabanına bir **MP4** dosyası eklemesini sağlayalım.

```
INSERT UserFiles(UserID, UserFile)
SELECT 1, BulkColumn
FROM OPENROWSET(BULK'C:\mehter.mp4',SINGLE_BLOB) AS UserVideoFile;
```

Sorgu sonucunda 1 kayıt etkilendiğine dair mesaj alacağız.

```
(1 row(s) affected)
```

UserID değeri 1 olan kullanıcının eklediği video dosyasını listeleyelim.

```
SELECT * FROM UserFiles WHERE UserID = 1;
```

	UserID	UserFile
1	1	0x0000001C667479704641434500000539697 36F6D6176633146414345000109AE6D6F6F760000006C6D7668640000000...

Şimdi de aynı kullanıcımız için bir resim dosyası ekleyelim.

```
INSERT UserFiles(UserID, UserFile)
SELECT 1, BulkColumn
FROM OPENROWSET(BULK'C:\dijibil_logo.png',SINGLE_BLOB) AS UserImageFile;
```

Eklediğimiz resim dosyası ile birlikte tüm kayıtları listeleyelim.

	UserID	UserFile
1	1	0x0000001C667479704641434500000539697 36F6D6176633146414345000109AE6D6F6F760000006C6D766864...
2	1	0x89504E470D0A1A0A0000000D49484452000000720000003008060000001BA4A1E8C0000000970485973000000...

Bu yöntem ile bir dokümanı ya da farklı bir dosyayı da veritabanına Binary olarak ekleyebilirsiniz.

Binary olarak eklediğimiz dosya kayıtlarının üzerinde güncelleme işlemi de gerçekleştirebiliriz.

mehter.mp4 dosyasını, *istiklal_marsi.mp4* ile değiştirerek güncelleyelim.

```
UPDATE UserFiles
SET UserFile = (
 SELECT BulkColumn
 FROM OPENROWSET(BULK 'C:\istiklal_marsi.mp4', SINGLE_BLOB) AS Files)
WHERE UserID = 1;
```

FILESTREAM

Bir önceki konumuzda, SQL Server'ın video, resim, doküman gibi, yapısal olmayan büyük verilerin tutulması için geliştirdiği **OPENROWSET** komutunun faydalarını inceledik. Bu özellik ile bir dosyayı, HardDisk'ten bağımsız olarak, veritabanı içerisinde saklaması, bir uygulamanın dosya depolama tekniklerini

kullanarak, veriyi daha güvenli bir ortamda depolaması için etkili ve kullanışlı bir yoldur.

Ancak bu yöntem ile veritabanı içerisinde dosya saklamak doğru bir yöntem değildir. Her ne kadar bu tür dosyaların saklanması mümkün ise de, SQL Server ve diğer tüm veritabanı yönetim sistemleri, metinsel veri depolama için tasarlanmıştır. Büyük veri dosyalarının veritabanında saklanması, veritabanı performansını olumsuz yönde etkileyeceği gibi, bazı kısıtlamalara da sahiptir. Örneğin; SQL Server bu tür dosyalar için, en fazla 2 GB veri depolama sınırına sahiptir.

Büyük verilerin depolanması için farklı bir çözüm yolu ise, dosyaların fiziki olarak dosya sistemi üzerinde tutularak, dosya yollarının veritabanında metinsel olarak tutulmasıdır. Bu yöntem ile performans sağlamak mümkün olacaktır. Ancak bir kullanıcının resim dosyasını dosya sistemi üzerinde tuttuğumuzu ve dosya sistemi üzerindeki bu dosyanın, adının ya da yolunun değiştirildiği olasılığını düşünelim. Bu durumda, SQL Server ile veri dosyası arasındaki bağ kopacak ve programsal olarak veritabanındaki dosya yolu üzerinden, dosya sistemindeki resim dosyasına erişilemez olunacaktır.

Bu iki yöntemin de dezavantajlarını göz önünde bulunduran Microsoft, SQL Server 2008 ile birlikte, **`FILESTREAM`** adı verilen yeni bir özellik geliştirdi.

`FILESTREAM` özelliği, bir dosyanın veritabanı MDF dosyaları üzerinde tutulmadan, dosya sistemi üzerinden tutulmasını sağladı. Buradaki en önemli fark, **`FILESTREAM`** ile saklanan dosyanın SQL Server üzerinden erişilebiliyor olmasıdır. Bu şekilde, dosyaların güvenliği, veri bütünlüğü, veritabanı şişmesinin engellenmesi, veritabanı performansının artması gibi birçok yönden olumlu yönde etkili bir özellik halini almıştır.

`FILESTREAM`, **`VARBINARY(MAX)`** veri tipinde veri tutabilen **BLOB** (*Binary Large Object*) bir özelliktir. **`FILESTREAM`**'e eklenen bir dosya, veritabanı MDF dosyasına değil, dosya sisteminde bir dosya olarak tutulur. Veritabanında ise disk üzerinde tutulan dosya ile ilişkili bir pointer tutulur. **`FILESTREAM`** özelliği ile birlikte, 2 GB dosya boyutu sınırlaması ortadan kalkar. HardDisk'in depolama üst sınırı kullanılır.

FILESTREAM Özelliğini Aktifleştirmek

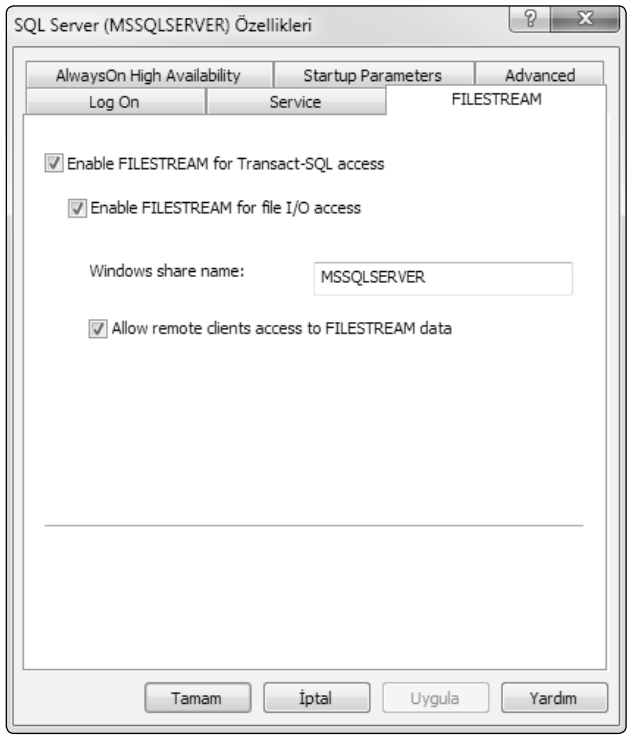
FILESTREAM özelliği; SQL Server kurulumunda varsayılan (*default*) olarak kapalı (*pasif*)'dır. Bu özelliği kurulum sırasında değiştirmediyseniz daha sonra da değiştirebilirsiniz.

Kurulum sonrasında **FILESTREAM** özelliğini aktifleştirmek için, aşağıdaki yolları takip edebilirsiniz.

- SQL Server Configuration Manager arayüzünü açmak için;

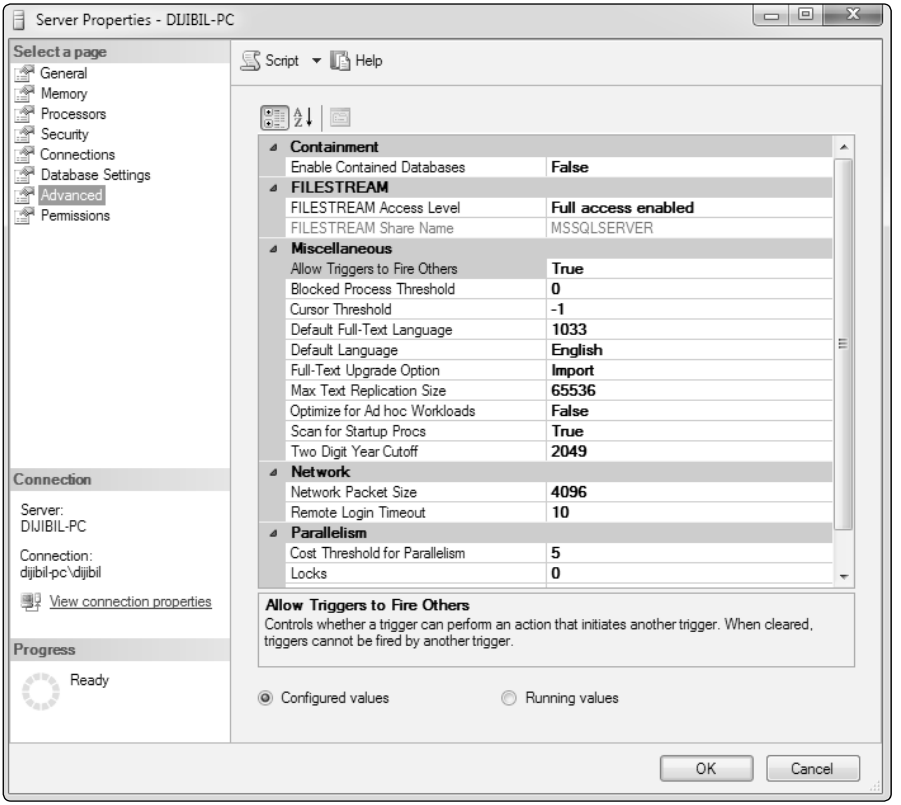
Programlar\Microsot SQL Server 2012\Configuration Tools\SQL Server Configuration Manager

Configuration Manager aracında **FILESTREAM** ayarlarını gerçekleştirmek için, sol bölümdeki **SQL Server Services** seçilmeli, daha sonra sağ kısımda **SQL Server (MSSQLSERVER)** simgesine sağ tıklanarak **Özellikler** menüsünde **FILESTREAM** tab'ına girilerek ilgili seçim kutuları seçilmeli.



Bu işlemleri gerçekleştirdikten sonra, **Filestream Access Level** ayarı düzenlenmelidir.

- **SSMS** ekranında, **Object Browser**'daki **Server Instance**'ı üzerine sağ tıklanarak **Properties** menüsü seçilir.
- Açılan pencerede, sol alandaki **Advanced** menüsü seçilir.
- Sağda listelenen **FILESTREAM** kısmındaki **Filestream Access Level** menüsünde erişim seviyesi belirlenir.



Aynı ayarların T-SQL ile gerçekleştirilmesi;

Yönetim araçları kullanılarak yapılan ayarlar T-SQL ile de şu şekilde gerçekleştirilebilir.

```
EXEC sp_configure filestream_access_level, 2
GO
RECONFIGURE
GO
```

sp_configure prosedürü iki parametre almaktadır.

- **1. parametre:** **filestream_access_level** olarak belirlendi. Sistemdeki ilgili işlemi yapacak ayarın ismi.
- **2. parametre:** 2 olarak belirlendi.

- 2. parametrede belirtilen değerlerin anlamları;

  0 değeri: Pasif

  1 değeri: Transact-SQL Access Enabled

  2 değeri: Full Access Enabled

Bu işlemler sonrasında, SQL Server servisini yeniden başlatarak bu hizmetin tam kullanıma girmesini sağlayabilirsiniz. Artık oluşturacağınız yeni veritabanlarında **Data** (*MDF*), **Log** (*LDF*) dosyalarının yanı sıra, **FILESTREAM Data Container** adlı bir klasör de oluşturulacaktır.

Mevcut Bir Veritabanında FILESTREAM Kullanmak

FILESTREAM işlemleri genel olarak, önceden var olan bir veritabanını revize etmek, veritabanı yeteneklerini geliştirmek gibi sebeplerden dolayı sonradan eklenir.

Biz de, kullandığımız **AdventureWorks** veritabanına **FILESTREAM** özelliği ekleyeceğiz.

Başlangıç seviyesindeki bölümlerde veritabanı oluşturulurken anlattığımız **FILEGROUP** konusunu bu örneğimizde de kullanacağız.

- Veritabanına **FILESTREAM** özelliği eklemek için, yeni bir **FILEGROUP** ekleyelim.

```
ALTER DATABASE AdventureWorks ADD
FILEGROUP FSGroup1 CONTAINS FILESTREAM;
```

- **FILEGROUP** üzerinde, **FILESTREAM** verilerinin tutulacağı **FILESTREAM Data Container** oluşturalım.

```
ALTER DATABASE AdventureWorks ADD FILE(
 NAME = FSGroupFile1,
 FILENAME = 'C:\Databases\AdventureWorks\ADWorksFS')
 TO FILEGROUP FSGroup1;
```

AdventureWorks veritabanına **FILESTREAM** özelliği kazandırma işlemimiz tamamlandı.

Şimdi, bir tablo oluşturarak bu özelliği test edelim.

```
CREATE TABLE ADVDocuments
(
 DocID  INT IDENTITY(1,1) PRIMARY KEY,
 DocGUID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL,
 DocFile VARBINARY(MAX) NOT NULL,
 DocDesc VARCHAR(500)
);
```

Tabloya bir kayıt ekleyelim.

```
INSERT INTO ADVDocuments(DocGUID, DocFile, DocDesc)
  VALUES(NEWID(),
  (SELECT *
   FROM OPENROWSET(BULK N'C:\kodlab.docx', SINGLE_BLOB) AS Docs),
'KodLab Tanıtım Dökümanı');
```

INSERT sorgumuz başarıyla çalışacak ve **docx** doküman dosyasını veritabanına Binary olarak ekleyecektir.

Bu sorguda kafa karışıklığı ve hata ile karşılaşmamanız için; Binary edilecek dosyayı belirttiğiniz **SELECT** sorgusundan sonra, başka bir parametre daha ekleyecekseniz, **SELECT** sorgusunu, takma ismi ile birlikte parantez içerisine almalısınız. Aksi halde sorgu hata üretecektir.

FILESTREAM Özelliği Aktif Edilmiş Bir Veritabanı Oluşturmak

FILESTREAM özelliğinin desteklendiği bir veritabanı oluşturalım.

Oluşturacağımız veritabanının bilgileri şu şekilde olacaktır.

- Veri dosyası adı: **DijiLabs.mdf**
- Log dosyası adı : **DijiLabs.ldf**
- FILESTREAM Klasörü: **DijiLabsFS**

Şimdi, veritabanımızı oluşturalım.

```
CREATE DATABASE DijiLabs
ON
PRIMARY(
 NAME = DijiLabsDB,
 FILENAME = 'C:\Databases\DijiLabs\DijiLabsDB.mdf'
),
FILEGROUP DijiLabsFS CONTAINS FILESTREAM(
 NAME = DijiLabsFS,
 FILENAME = 'C:\Databases\DijiLabs\DijiLabsFS'
)
LOG ON(
 NAME = DijiLabsLOG,
 FILENAME = 'C:\Databases\DijiLabs\DijiLabsLOG.ldf'
);
```

Veritabanı oluşturma konusunu işlediğimiz bölümdeki bilgileri hatırlayın. Şimdi oluşturduğumuz veritabanında farklı olan tek şey, bir **FILEGROUP** olarak eklenen **DijiLabsFS** dosya grubudur.

Veritabanının sağlıklı bir şekilde oluşturulabilmesi için **C:** dizini içerisinde **Databases** klasörü ve bu klasör içerisinde **DijiLabs** adında, veritabanımız ile ilgili tüm dosyaların ve klasörlerin bulunacağı özel bir klasör oluşturduk.

DijiLabsFS adındaki **FILESTREAM** klasörü, kendi içerisinde dosya ve klasör oluşturacağı için, **DijiLabsFS** klasörünü biz oluşturmadık. Hazırladığımız script çalışırken, bu klasör otomatik olarak **DijiLabs** klasörü içerisinde otomatik olarak oluşturulacaktır.

DijiLabsFS klasörü içerisinde **filestream.hdr** adında bir dosya ve **$FSLOG** adında bir klasör oluşturuldu.

- **filestream.hdr:** `FILESTREAM` özelliği ile saklanacak dosyalar için oluşacak metadata'ların depolandığı dosyadır.
- **$FSLOG:** FILESTREAM ile ilgili log'ların depolandığı klasördür.

Sütunları Oluşturmak

`FILESTREAM` veri tutabilen veritabanı oluşturduk. Ancak tek başına bu yeterli değildir. Tablo ve sütun seviyesinde bazı özelliklerin de oluşturulması gerekir.

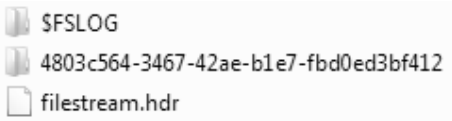
DijiLabs veritabanımızda, DİJİBİL'de AR-GE görevlisi olarak çalışan personellerin bilgilerini tutacağız. Bunun için bir tablo oluşturalım.

```
CREATE TABLE Person(
 PersonID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
 FirstName VARCHAR(30),
 LastName VARCHAR(30),
 Email  VARCHAR(50),
 PImage VARBINARY(MAX) FILESTREAM NULL
);
```

`FILESTREAM` özelliği ile veri tutulacak bir tabloda `UNIQUE` ve `ROWGUIDCOL` özelliklerine sahip bir `UNIQUEIDENTIFIER` veri tipinde sütun bulunması zorunludur. Bu özelliklerin oluşturulmadığı bir tablo oluşturulmaya çalışıldığında hata verecektir.

Şimdi, `Person` tablosunu oluşturduktan sonra **DijiLabsFS** klasörüne tekrar bakalım.

GUID ile isimlendirilmiş bir klasör oluşturulduğunu görüyoruz.



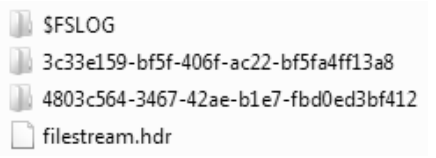
Bir tabloda birden fazla `FILESTREAM` özelliğine sahip sütun bulunabilir.

Person1 adında farklı bir tablo daha oluşturalım.

```
CREATE TABLE Person1(
 PersonID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
 FirstName VARCHAR(30),
 LastName VARCHAR(30),
 Email  VARCHAR(50),
 PImage VARBINARY(MAX) FILESTREAM NULL,
 PImage1 VARBINARY(MAX) FILESTREAM NULL
);
```

FILESTREAM özelliğini destekleyen birden fazla sütunlu bir tablo oluşturduk.

Person1 tablosunu oluşturduktan sonra **DijiLabsFS** klasörüne tekrar bakın. **GUID** ile isimlendirilmiş yeni bir klasör daha oluşturulduğunu görebilirsiniz.



Veri Eklemek

DijiLabs veritabanında oluşturulan ve **FILESTREAM** desteği olan **Person** tablosuna bir kayıt ekleyelim.

```
INSERT INTO Person(PersonID, FirstName, LastName, Email, PImage)
SELECT NEWID(),
  'Cihan',
  'ÖZHAN',
  'cihan.ozhan@dijibil.com',
  CAST(BulkColumn AS VARBINARY(MAX))
  FROM OPENROWSET(BULK 'C:\dijibil.png',SINGLE_BLOB) AS ImageData;
```

Veri Seçmek

FILESTREAM özelliği ile NTFS dosya sisteminde depolanan verilerin **SELECT** ile seçilmesi işlemi de basittir.

Person tablosuna eklediğimiz kaydı listeleyelim.

```
SELECT * FROM Person;
```

Veri Güncellemek

FILESTREAM özelliği ile veritabanına eklediğimiz bir kaydı güncelleyelim.

Güncelleme işlemi basittir. Ancak dikkat edilmesi gereken şey; **WHERE** koşulu kullanarak, sadece istediğiniz veri ya da verilerin güncellemeye tabi tutulmasını sağlamaktır. **WHERE** koşulu kullanmadığınız takdirde tüm Binary veri dosyalarınız aynı dosya ile güncellenecektir.

Person tablosunda bir kaydı güncelleyelim.

```
UPDATE Person
SET PImage = (
SELECT BulkColumn
FROM OPENROWSET(BULK 'C:\mehter.mp4', SINGLE_BLOB) AS VideoData)
WHERE PersonID = '96C735D4-60B4-4C0C-B108-A0185F8BD5E2';
```

Güncelleme işleminde, **WHERE** koşulunda **PersonID** sütununa belirttiğim değer bir GUID olduğu için, sizin bilgisayarınızda, sizin eklediğiniz kayıtlarda farklı olacaktır. Bu nedenle güncelleme yapmak istediğiniz kaydın, **GUID** değerini kopyalayarak sorgumda kullandığım ('**96C735D4-60B4-4C0C-B108-A0185F8BD5E2**') değer yerine kullanın.

Person tablosunu oluştururken **PersonID** sütunu için farklı bazı özellikler kullandığımızı belirtmiştik. Bu özelliklerden biri de **ROWGUIDCOL** idi. Ve **INSERT** ile veri eklerken de **NEWID()** fonksiyonu ile bu sütuna **GUID** tipinde bir değer ekliyorduk.

Şimdi **WHERE** ile tek bir kayıt güncelleme işlemi gerçekleştirebilmek için bu GUID değere ihtiyacımız var. En kısa yol olarak, SSMS'de bir **SELECT** sorgusu çalıştırın ve listelenen sonuçlardan **PersonID** üzerinde ilgili kaydın hücresine sağ tıklayarak kopyalayın.

Kopyaladığınız **GUID** değerini, **UPDATE** sorgusundaki **PersonID** sütununda eşittir (=) işaretinden sonra, tek tırnaklar (' ') içerisinde yazmalısınız. Aksi halde hata ile karşılaşırsınız.

GUID Değere Sahip Sütun Oluşturulurken Dikkat Edilmesi Gerekenler

Bir tabloda **GUID** değer içeren sütun kullanmanız gerekiyor ise, bu değeri tek başına **ID** değerlerini depoladığınız sütun olarak kullanmamalısınız. Bu örnekte kullanılan **PersonID** değeri, aslında otomatik artan bir nümerik değer olması gerekirdi. **GUID** olarak kullanılacak sütunu ise, ayrı bir sütun olarak oluşturulmalıydı.

Güncelleme işleminde de fark ettiğiniz gibi, **GUID** olan bir sütun değerinin **ID** olarak kullanılmasının bazı sıkıntıları var.

Person tablosu şu şekilde olabilirdi.

```
CREATE TABLE Person(
 PersonID INT NOT NULL IDENTITY(1,1),
 PersonGUID UNIQUEIDENTIFIER ROWGUIDCOL UNIQUE NOT NULL,
 FirstName VARCHAR(30),
 LastName VARCHAR(30),
 Email  VARCHAR(50),
 PImage VARBINARY(MAX) FILESTREAM NULL
);
```

Bu örnekte **PersonID** değeri ile **PersonGUID** değeri birbirinden bağımsızdır. **PersonID**, otomatik olarak artar ve **PersonGUID** ise **UNIQUE**, yani benzersiz değerlere sahiptir. Ancak, normal sorgulamalarda geliştirici olarak siz **PersonID** sütununu kullanmalısınız.

Konu hakkında tecrübe edindikten sonra konuyu açıklamak istediğim için, tablo oluşturma sırasında değil, şuan anlatmayı uygun gördüm.

Veri Silmek

FILESTREAM tablolarda veri silme işlemi T-SQL tarafında farklı değildir. Ancak arka planda farklı işlemler gerçekleşir. Dosya sistemini kullanan ve dosyaları yöneten bir özelliği olduğu için, bir kayıt silindiğinde, NTFS dosya sistemindeki veritabanı kaydı ile ilişkili dosyayı silmek için bir **Garbage Collector** (*çöp toplayıcı*) görevlendirilir. Belli aralıklarla sistemi kontrol eden bu çöp toplayıcı, zamanı geldiğinde hafızadaki bu dosyaları siler.

Oluşturduğum tabloda, güncelleme işleminde kullandığım GUID değerine sahip kaydı sileceğim.

```
DELETE FROM Person WHERE PersonID = '96C735D4-60B4-4C0C-B108-
A0185F8BD5E2';
```

DELETE sorgusunda bile, **PersonID** değeri ile **GUID** değer tutacak farklı bir sütun oluşturmak gerektiğini fark etmiş olmalısınız.

Verileri Gruplamak ve Özetlemek

Bir veritabanı yönetim sistemini kullanmanın en önemli amacı veriyi saklamaktan ziyade, saklanan veriyi performanslı bir şekilde işlemek ve yönetmektir. Ek özel gereksinimler ise oldukça fazladır. Bu gereksinimlerin başında ise; veriler üzerinde istatistik ve raporlama gibi ihtiyaçları karşılayacak programsal alt yapıya sahip olmak vardır. Bu işlemleri genellemek için gruplama terimi kullanılır.

SQL Server gruplama işlemlerinde oldukça yeteneklidir. Birçok fonksiyon ve komut yapısı ile gruplama ve özetleme işlemlerini kolaylaştırır.

Group By

Group By deyimi, tabloyu veya birlikte sorgulanan tabloları, gruplara bölmek için kullanılır. Genel olarak grup başına ayrı istatistikler ürettirmek, hesaplamalar yapmak için kullanılır.

Söz Dizimi:

```
SELECT sutun_ad, gruplamalı-fonksiyon(sutun_ad)
FROM tablo_ad
WHERE şartlar
GROUP BY sutun_ad;
```

Group By deyiminin en yaygın kullanıldığı örnek, satış ya da sipariş toplamını hesaplama işlemidir.

GROUP BY deyiminin amacını kavrayabilmek için öncelikle sorunu tespit etmeliyiz. Burada çözülmesi gereken konu siparişlerin gruplanması, yani bir üründen kaç sipariş alındığını bulmaktır.

Bunu en temel haliyle basit bir **WHERE** koşulu oluşturarak bulabiliriz.

```
SELECT
 SalesOrderID, OrderQty
FROM
 Sales.SalesOrderDetail
WHERE
 SalesOrderID BETWEEN 43670 AND 43680;
```

	SalesOrderID	OrderQty
1	43670	1
2	43670	2
3	43670	2
4	43670	1
5	43671	1
6	43671	2
7	43671	1

Bu sorgumuz ile 10 sipariş sorgulayarak listeledik. Ancak sorgu sonucunda 93 satır kayıt listelendi. Bu işlemde bizim asıl isteğimiz bu siparişlerin tek tek getirilmesi değil, her siparişte hangi ve kaç ürünün alındığını görebilmekti. O halde bu sorgu dolaylı olarak işe yarasa da bizim istediğimiz özellikleri karşılamıyor.

Bizim isteğimizi karşılayacak işlem verileri gruplamak ile gerçekleşebilir. Bunun için **GROUP BY** komutunu kullanarak bu sorguyu tekrar çalıştıralım.

```
SELECT
 SalesOrderID AS [Sipariş NO],
 SUM(OrderQty) AS [Sipariş Adet]
FROM
 Sales.SalesOrderDetail
WHERE
 SalesOrderID BETWEEN 43670 AND 43680
GROUP BY
 SalesOrderID;
```

	Sipariş NO	Sipariş Adet
1	43670	6
2	43671	17
3	43672	9
4	43673	20
5	43674	3
6	43675	22
7	43676	12

Bu örneğimizde ise **Sales.SalesOrderDetail** tablosundaki verileri kullanarak satış toplamını hesaplayalım.

```
SELECT          55.PNG
 ProductID,
 COUNT(ProductID) AS [ToplamSatılanUrun]
FROM
 Sales.SalesOrderDetail
GROUP BY
 ProductID;
```

	ProductID	ToplamSatılanUrun
1	707	3083
2	708	3007
3	709	188
4	710	44
5	711	3090
6	712	3382
7	713	429

Bu sorgumuza **WHERE** koşulu ekleyerek sadece bir ürünün hesaplamasını yapmak istersek;

```
SELECT
 ProductID,
 COUNT(ProductID) AS [ToplamSatılanUrun]
FROM
 Sales.SalesOrderDetail
WHERE
 ProductID = 707
GROUP BY
 ProductID;
```

	ProductID	ToplamSatılanUrun
1	707	3083

WHERE filtresi ile **707 ProductID** değerine sahip kaydı tek başına listeledik.

Listelediğimiz satılan ürünlerin toplamı, '*en çok satılan*' ve '*en az satılan*' olarak bölümlendirmek istersek;

En çok satılanların listelenmesi

```
SELECT
 ProductID,
 COUNT(ProductID) AS [ToplamSatılanUrun]
FROM
 Sales.SalesOrderDetail
GROUP BY ProductID
ORDER BY [ToplamSatılanUrun] DESC;
```

	ProductID	ToplamSatılanUrun
1	870	4688
2	712	3382
3	873	3354
4	921	3095
5	711	3090
6	707	3083
7	708	3007

En az satılanları listelemek için tek yapmanız gereken **DESC** yazan kısma **ASC** yazmaktır.

Group By deyimi; listeleme işlemleri için değil, hesaplama ve istatistiksel veriler üretmek için kullanılır. Sadece listeleme işlemi gerçekleştirilmek isteniyorsa **Group By** yerine **Order By** kullanılması daha doğru olacaktır. Ancak gruplama yapılmış veri üzerinde de **Order By** kullanarak listeleme işlemi yapabilirsiniz.

Gruplama işlemi sadece tek sütuna göre yapılmak zorunda değildir. Gruplama yapmak istenen diğer sütunları, **SELECT** sorgusunda yazarak ve **Group By** deyiminde de aralarına virgül koyarak, birden fazla gruplama tek sorgu ile gerçekleştirebilir. **SELECT** sorgusunda belirtilen sütunlar **GROUP BY** koşulunda yer almalı ya da aggregate olmalıdır.

Aggregate konusu **Gruplamalı Fonksiyonlar** bölümünde detaylıca incelenecektir.

Group By All

Group By All operatörü **Group By** ile aynı şekilde çalışır. **Group By All** operatörünün tek farkı, gruplama işlemini tüm kayıtlar üzerinde yapıyor olmasıdır. **WHERE** ile bir koşul oluşturulsa bile, bu operatör oluşturulan koşulu dikkate almadan tüm kayıtları listeleyecektir.

Bu iki gruplama özelliği arasındaki farkı anlamak için ikisini de daha önce hazırladığımız örnek üzerinde inceleyelim.

Satılan ürünlerin toplamını listelediğimiz sorgumuzda bir **WHERE** koşulu uyguladık.

```
SELECT          58.PNG
 ProductID,
 COUNT(ProductID) AS [ToplamSatılanUrun]
FROM
 Sales.SalesOrderDetail
WHERE ProductID < 800
GROUP BY ProductID;
```

	ProductID	ToplamSatılanUrun
1	707	3083
2	708	3007
3	709	188
4	710	44
5	711	3090
6	712	3382
7	713	429

Bu sorgumuzun sonucunda 83 kayıt listelenecektir.

Şimdi aynı sorguyu **Group By All** ile hazırlayalım.

```
SELECT          59.PNG
 ProductID,
 COUNT(ProductID) AS [ToplamSatılanUrun]
FROM
 Sales.SalesOrderDetail
WHERE ProductID < 800
GROUP BY ALL ProductID;
```

	ProductID	ToplamSatılanUrun
1	707	3083
2	708	3007
3	709	188
4	710	44
5	711	3090
6	712	3382
7	713	429

Bu sorgumuzda ise 266 kayıt listelendi. Bunun sebebi; açıklamamızda da belirttiğimiz gibi **Group By All** deyiminin tüm kayıtlar üzerinde gruplama yapıyor olmasıdır.

HAVING İLE GRUPLAMALAR ÜSTÜNDE ŞART KOŞMAK

Şart oluşturmak denilince ilk akla gelen deyim **WHERE** olur. Satırlar üzerinde **WHERE** ile şart oluşturulabilir. Ancak, satırları gruplara ayırdıktan sonra, bu gruplardan şartları sağlayanları listeleyip, şartı sağlamayanların sonuçta yer almasını önlemek için **HAVING** deyimi kullanılır.

HAVING koşulu sorguda sadece **GROUP BY** koşulu varsa kullanılır. **GROUP BY** koşulu olmayan sorguda kullanılan **HAVING** koşulu **WHERE** ile aynı anlama sahiptir.

Kayıt filtrelemeleri için gruplamalı fonksiyonlar kullanılacaksa **HAVING** deyimi ile belirtilmelidir.

Söz Dizimi:

```
SELECT sutun_ismi1, Gruplamali_Fonksiyon(sutun_ismi)
FROM tablo_ismi
WHERE sartlar
GROUP BY sutun_ismi1
HAVING Gruplamali_Fonksiyon(sutun_ismi)
[ORDER BY siralayici]
```

Her bir ürün için kaç adet satıldığını hesaplayalım.

```
SELECT ProductID, COUNT(ProductID) ToplamSatilanUrun
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING COUNT(ProductID) > 500
ORDER BY ToplamSatilanUrun DESC;
```

	ProductID	ToplamSatilanUrun
1	870	4688
2	712	3382
3	873	3354
4	921	3095
5	711	3090
6	707	3083
7	708	3007

Gruplamalı Fonksiyonlar (Aggregate Functions)

Gruplamalı fonksiyonlar SQL Server'da gruplama işlemlerini yapmak için var olan fonksiyonlardır. Tablolarda oluşturulan veri gruplarına ait verileri özetleyen fonksiyonlardır. Bir sorgunun gruplara ayrılarak her bir grup için özet bilgiler alınması amacıyla kullanılırlar.

AVG Fonksiyonu

Average'ın kısaltması olan **AVG** fonksiyonu verilen değerlerin ortalamasını hesaplar.

Söz Dizimi:

```
SELECT AVG(alan_ad)
FROM tablo_ad
```

Production.Product tablosunda fiyatları tutan **StandardCost** sütununun **AVG** ile ortalamasını alalım.

```
SELECT
 AVG(StandardCost)
FROM
 Production.Product;
```

	(No column name)
1	258,0908

Sorgularımızda genel olarak iki çeşit hesaplama yöntemi kullanmak isteriz.

- Tüm kayıtlar üzerinde hesaplama (**ALL**)
- Benzersiz kayıtlar üzerinde hesaplama (**DISTINCT**)

AVG() fonksiyonu da bu isteklere cevap verebilmektedir.

Varsayılan kullanım olarak **AVG(sutun_ad)** kullanımı geçerli olsa da bunun SQL Server mimarisinde gerçek karşılığı **AVG(ALL sutun_ad)**'dır. Buna **AVG()**'nin varsayılan kullanımı diyebiliriz.

Bu durumda yukarıdaki örnek kullanım şekliyle aşağıdaki kullanım SQL Server mimarisine göre aynıdır.

```
SELECT
 AVG(ALL StandardCost)
FROM
 Production.Product;
```

	(No column name)
1	258,0908

AVG(ALL sutun_ad) kullanımı ile **AVG()** kullanımı tekrar eden ya da etmeyen tüm kayıtlar üzerinde hesaplama işlemi yaparak bize ortalama döndürür. Peki ya tekrar etmeyen (her aynı kayıttan bir adet) kayıtlar üzerinde hesaplama işlemi yapmak istersek?

Bu durumda **AVG(DISTINCT sutun_ad)** kullanımı işimizi görecektir. **DISTINCT** ifadesi kelime olarak FARKLI anlamına gelir. Buradan da anlayacağımız gibi bize farklı olan kayıtları getirmek için tasarlanmıştır.

Hemen bir örnek yaparak **AVG(DISTINCT sutun_ad)** kullanımını inceleyelim.

```
SELECT
 AVG(DISTINCT StandardCost) AS [FARKLI]
FROM
 Production.Product;
```

	FARKLI
1	266,2329

StandardCost sütunu üzerinde çalıştırdığımız **DISTINCT** sorgusunun nasıl çalıştığını bir örnek ile inceleyelim.

```
SELECT
 Name, StandardCost
FROM
 Production.Product;
```

	Name	StandardCost
1	Adjustable Race	0,00
2	Bearing Ball	0,00
3	BB Ball Bearing	0,00
4	Headset Ball Bearings	0,00
5	Blade	0,00
6	LL Crankarm	0,00
7	ML Crankarm	0,00

Sağ taraftaki kayıtlarda görüldüğü gibi;

- 187, 190 ve 193. kayıtların değeri 98.77
- 188, 191, 194. kayıtların değeri 108.99
- 189, 192, 195. kayıtların değeriyse 145.87'dir.

Bu kayıtlara göre sorgumuzu **ALL** ile hazırladığımızda yukarıdaki tekrarlanan tüm kayıtları hesaplama işlemine tabi tutacaktır. Ancak sorgumuz **DISTINCT** ile hazırlandığında bu tekrar eden kayıtlardan sadece birer tanesini hesaplayacaktır. Bu durumda 98.77, 108.99 ve 145.87 **StandardCost** değerine sahip kayıtlardan sadece birer tane olduğu varsayılarak bir sonuç üretilecektir.

SUM Fonksiyonu

Toplama işlemi yapan bir fonksiyondur. Nümerik bir alandaki tüm kayıtların toplamını verir.

Söz Dizimi:

```
SELECT SUM(sutun_ad)
FROM tablo_ad
```

Production.Product tablomuzda **SafetyStockLevel** ve **ListPrice** sütunlarını kullanarak **SUM()** fonksiyonunu hesaplama işleminde kullanalım.

```
SELECT
 SUM(ListPrice) AS Total_ListPrice,
 SUM(SafetyStockLevel) AS [Safety Stock]
FROM Production.Product;
```

	Total_ListPrice	Safety Stock
1	251768,3491	270716

SUM() fonksiyonunda da diğer bazı fonksiyonlarda olduğu gibi parametre olarak **ALL** ve **DISTINCT** kullanımı mümkündür. Yukarıdaki örneğimizi şimdi parametreli olarak değiştirelim.

```
SELECT
 SUM(DISTINCT ListPrice) AS Total_ListPrice,
 SUM(ALL SafetyStockLevel) AS [Safety Stock]
FROM Production.Product;
```

	Total_ListPrice	Safety Stock
1	57590,7762	270716

Bu sorgumuzda **ListPrice** sütunu için **DISTINCT** uyguladık. Yani aynı **ListPrice** değerine sahip kayıtların tek kayıt olarak algılanması ve içlerinden bir tanesinin hesaplamaya tabi tutulmasını sağladık.

SafetyStockLevel sütununda ise varsayılan kullanım olan **SUM()** parametresiz kullanımı ile aynı işlevi görmektedir. Yani tüm kayıtları ayrım yapmadan toplayarak hesaplar.

Bir **SELECT** sorgusunda birden fazla **SUM()** fonksiyonunun kullanılabildiğini öğrendik. Peki **SUM()** fonksiyonu ile birlikte herhangi bir sütun adı kullanabilir miyiz?

```
SELECT Color, SUM(ListPrice), SUM(StandardCost) FROM Production.Product;
```

Bu sorguyu çalıştırmak istediğimizde aşağıdaki gibi bir hata mesajıyla karşılaşırız.

```
Column 'Production.Product.Color' is invalid in the select list
because it is not contained in either an aggregate function or the
GROUP BY clause.
```

Bunun anlamı **SUM()** fonksiyonu ile herhangi bir sütunu ek işlem yapmadan kullanamayacak olmamızdır.

SUM() ile farklı bir sütun kullanmamız için sorgumuza **GROUP BY** deyimini eklememiz gerekmektedir.

Production.Product tablomuzda **Color** sütunu değeri **NULL** olmayan, **ListPrice** sütunu değeri 0.00 olmayan ve adı **Mountain** ile başlayan kayıtların **ListPrice** ve **StandardCost** sütunlarını **SUM()** fonksiyonu ile toplayıp, **Color** sütununa göre gruplayıp, **Color** sütununa göre listelemek istiyoruz.

```
SELECT
 Color, SUM(ListPrice),
 SUM(StandardCost)
FROM
 Production.Product
WHERE
     Color IS NOT NULL
 AND ListPrice != 0.00
     AND Name LIKE 'Mountain%'
GROUP BY Color
ORDER BY Color;
```

	Color	(No column name)	(No column name)
1	Black	31443,253	15214,9616
2	Silver	30362,4382	14665,6792
3	White	21,7998	6,7926

COUNT Fonksiyonu

Kayıt sayıcı olarak nitelendirebileceğimiz, bir alanda bulunan kayıtları sayarak sonucu döndüren fonksiyondur.

Söz Dizimi:

```
SELECT COUNT(*)
FROM tablo_ad
```

Production.Product tablosundaki kayıtları, yani ürünleri sayalım.

```
SELECT COUNT(*)       68.PNG
FROM Production.Product;
```

	(No column name)
1	505

***** ile sütun belirtmek zorunda değiliz. Bir sütun adı vererek de toplama işlemi gerçekleştirilebilir.

Production.Product tablosundaki **Name** sütununu kullanarak ürünleri sayalım.

```
SELECT COUNT(Name)      69.PNG
FROM Production.Product;
```

	(No column name)
1	505

Her zaman tablodaki kayıtların tamamını sayma durumu söz konusu olmayabilir. Bazen istediğimiz özellikteki kayıtların adedini öğrenmek isteyebiliriz. Bu işlem için **WHERE** deyimini kullanırız.

Ürün adı '**A**' ile başlayan kayıtların tamamını getirmek için aşağıdaki sorguyu hazırladık.

```
SELECT * FROM Production.Product WHERE Name LIKE 'A%';
```

	ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	1	Adjustable Race	AR-5381	0	0	NULL	1000	750	0,00	0,00
2	461	Advanced SQL Server	LR-2398	0	0	Silver	1000	750	0,00	0,00
3	712	AWC Logo Cap	CA-1098	0	1	Multi	4	3	6,9223	10,3148
4	879	All-Purpose Bike Stand	ST-1401	0	1	NULL	4	3	59,466	182,4305

Görüldüğü gibi **Production.Product** tablomuzda 4 kayıt bize sonuç olarak döndü.

Bu kayıtların tüm sütun değerlerini getirmek yerine sadece sayısını almak istiyorsak, aşağıdaki sorguyu kullanabiliriz.

```
SELECT COUNT(*) FROM Production.Product WHERE Name LIKE 'A%';
```

	(No column name)
1	4

Burada kullanılan asteriks (**`*`**) işareti herhangi bir sütununda değer olan tüm girilmiş kayıtları toplar. Sütun bazlı **`COUNT(sutun_ad)`** kullanım gerçekleştirilirse **`NULL`** kayıtlar hesap dışı tutulacaktır. Ancak **`COUNT(*)`** işlemine tabi tutulan tablodaki **`NULL`** olanlar dahil tüm kayıtlar hesaplanacaktır.

MAX Fonksiyonu

Bir alanda bulunan kayıtların arasındaki nümerik ya da alfabetik en büyük değeri bulur.

Söz Dizimi:

```
SELECT MAX(sutun_ad)
FROM tablo_ad
```

Bu fonksiyonun kullanımını kavramanın en kolay yolu bir identity sütun üzerinden test etmek olacaktır.

`Production.Product` tablosundaki **`ProductID`** sütununa **`MAX()`** fonksiyonu ile en yüksek değer bulma işlemi gerçekleştirelim.

```
SELECT
 MAX(ProductID) AS [En Buyuk]
FROM
 Production.Product;
```

	En Buyuk
1	1004

MAX() fonksiyonu alt sorgular (sub query) ile de kullanabiliriz.

```
SELECT ProductID, Name
FROM
 Production.Product
WHERE
 ProductID = (SELECT
   MAX(ProductID)
   FROM Production.Product);
```

	ProductID	Name
1	1004	% 20 indirimli ürün

Alt sorguları ileriki bölümlerde detaylarıyla inceleyeceğiz.

Bu işlemde **ProductID = (...)** kısmında verilen değer **Production.Product** tablosundaki en yüksek değerdir. Bendeki **AdventureWorks** versiyonunun ilgili tablosundaki değer 999'dur.

Buraya kadar nümerik işlemler üzerinde **MAX()** fonksiyonu kullanımını gerçekleştirdik. Şimdi açıklamamızda bahsettiğimiz alfabetik işlemlerdeki çalışma mantığını inceleyelim.

Tahmin edersiniz ki alfabetik ya da nümerik olsun, karakterlerin bilgisayar biliminde bir sıralaması vardır. Bu sıralama alfabetik olarak A-Z, nümerik olarak ise 0-9 şeklinde hesaplanmaktadır. Burada bir nümerik değerin en büyüğünü istersek otomatik olarak 9 ya da 9'a en yakın olan karakter seçilecektir. Karakter bazında düşünürsek **DESCENDING(DESC)** bir sıralamada da nasıl Z'den başlayarak tersine bir sıralama işlemi gerçekleşiyorsa, **ASCENDING(ASC)** işleminde nasıl A'dan başlayan bir sıralama işlemi gerçekleşiyor ise, aynı şekilde **MAX()** fonksiyonunda da karakter işlemlerinde en yüksek, yani Z'den A'ya doğru bir sıralama işlemine göre en yukarıda bulunan kayıt getirilir.

Şimdi **Production.Product** tablomuzda **Name** sütunumuzu **MAX()** işlemine tabi tutalım.

```
SELECT MAX(Name) FROM Production.Product;
```

Görüldüğü gibi tablomuzdaki kayıtlardan Z'den A'ya en büyük karakter olan 'W' ile başlayan kayıt getirilmiştir.

	(No column name)
1	Women's Tights, S

Eğer ilk karakter aynı olan çok kayıt varsa 2, 3, 4 şeklinde son karaktere kadar kontrol etmeye devam edilecektir.

MIN Fonksiyonu

Bir alanda bulunan kayıtların arasındaki nümerik ya da alfabetik en küçük değeri bulur.

Söz Dizimi:

```
SELECT MIN(sutun_ad)
FROM tablo_ad
```

Bu fonksiyonun kullanımını kavramanın en kolay yolu bir **IDENTITY** sütun üzerinden test etmek olacaktır.

Production.Product tablosundaki **ProductID** sütununa **MIN()** fonksiyonu ile en küçük değer bulma işlemi gerçekleştirelim.

```
SELECT
 MIN(ProductID) AS [En Kucuk]
FROM
 Production.Product;
```

	En Kucuk
1	1

MIN() fonksiyonu alt sorgular (sub query) ile de kullanabiliriz.

```
SELECT ProductID, Name      76.PNG
FROM
 Production.Product
WHERE
 ProductID = (SELECT
   MIN(ProductID)
   FROM Production.Product);
```

	ProductID	Name
1	1	Adjustable Race

Bu işlemde **ProductID = (...)** kısmında verilen değer **Production.Product** tablosundaki en küçük değerdir. Benim **AdventureWorks** veritabanımın versiyonunun ilgili tablosundaki değer 1' dir.

Buraya kadar nümerik işlemler üzerinde **MIN()** fonksiyonu kullanımını gerçekleştirdik. Şimdi açıklamamızda bahsettiğimiz alfabetik işlemlerdeki çalışma mantığını inceleyelim.

MAX() fonksiyonu için gerekli karakter bazlı çalışma mantığı, **MIN()** fonksiyonu için de geçerlidir. Bu sefer Z'den A'ya değil, A'dan Z'ye yani küçükten büyüğe doğru sıralama gerçekleştirilerek, ilk olarak A karakterine bakmak kaydı ile A ya da A'ya en yakın kayıt getirilecektir.

Şimdi **Production.Product** tablomuzda Name sütunumuzu **MIN()** işlemine tabi tutalım.

```
SELECT MIN(Name) FROM Production.Product;
```

Görüldüğü gibi tablomuzdaki kayıtlardan A'dan Z'ye en küçük karakter olan A ile başlayan kayıt getirilmiştir.

	(No column name)
1	% 20 indirimli ürün

Eğer ilk karakter aynı olan çok kayıt varsa 2, 3, 4 şeklinde son karaktere kadar kontrol etmeye devam edilecektir.

Gruplanmış Verileri Özetlemek

Gruplama işlemleri, verileri belli gruplara ayırmak ve ayrılan gruplar hakkında belli değerleri bulmak için kullanılır. Bu bölümde, **GROUP BY** deyimi ile gruplanan verilerin üzerinde istatistiksel özetler elde edilmesini sağlayan parametreleri inceleyeceğiz.

CUBE

Cube deyimi, gruplanan veriler üzerinde kullanılır ve gruplama işlemini küpe dönüştürür. Küp, veri analizi ile ilgili bir terimdir. **GROUP BY** ile gruplara ayrılan veriler üstünde bütün ilişkileri göstermek için çeşitli sonuçlar elde edilmesini sağlar.

Adventure Works firmasında hangi tüm departmanların çalışan sayısını hesaplayalım.

```
SELECT D.Name, COUNT(*) AS [Çalışan Sayısı]
FROM HumanResources.EmployeeDepartmentHistory AS EDH
INNER JOIN HumanResources.Department AS D
ON EDH.DepartmentID = D.DepartmentID
GROUP BY CUBE(D.Name);
```

	Name	Çalışan Sayısı
1	Document Control	5
2	Engineering	7
3	Executive	2
4	Facilities and Maintenance	7
5	Finance	11
6	Human Resources	6
7	Information Services	10

Adventure Works firmasında 6 ya da daha az çalışanı bulunan departmanları listeleyelim.

```
SELECT D.Name, COUNT(*) AS [Çalışan Sayısı]
FROM HumanResources.EmployeeDepartmentHistory AS EDH
INNER JOIN HumanResources.Department AS D
ON EDH.DepartmentID = D.DepartmentID
GROUP BY CUBE(D.Name)
HAVING COUNT(EDH.DepartmentID) <= 6;
```

	Name	Çalışan Sayısı
1	Document Control	5
2	Executive	2
3	Human Resources	6
4	Production Control	6
5	Research and Development	4
6	Shipping and Receiving	6
7	Tool Design	4

CUBE, aynı zamanda satır sonu hesaplama işlemi için de kullanılabilir.

Aşağıdaki **GROUP BY** işleminde 3 kayıt listelenecektir.

```
SELECT i.Shelf, SUM(i.Quantity) Total
FROM Production.ProductInventory AS i
WHERE i.Shelf IN('A','B','C')
GROUP BY i.Shelf;
```

	Shelf	Total
1	A	14655
2	B	9823
3	C	16281

Bu kayıtların satır sonu toplamını almak için;

```
SELECT i.Shelf, SUM(i.Quantity) Total
FROM Production.ProductInventory AS i
WHERE i.Shelf IN('A','B','C','D')
GROUP BY CUBE(i.Shelf);
```

	Shelf	Total
1	A	14655
2	B	9823
3	C	16281
4	D	16768
5	NULL	57527

ROLLUP

ROLLUP deyimi, alt toplam ve genel toplam hesaplamaları için kullanılır. **ROLLUP** deyimi, sadece içten dışa doğru sütunlara ait toplamları bularak ilerler.

Bir **ROLLUP** örneği hazırlayalım.

```
CREATE TABLE tbPopulation (
 Category VARCHAR(100),
 SubCategory VARCHAR(100),
 BookName VARCHAR(100),
 [Population (in Millions)] INT
);
```

Örnek kayıtlar ekleyelim.

	Category	SubCategory	BookName	Population
1	Software	Java	Ileri Seviye Java Programlama	8
2	Software	PHP	PHP 6	6
3	Software	PHP	PHP 6 ile E-Ticaret Uygulamasi Gelistirme	6
4	Software	Android	Ileri Seviye Android Programlama	9
5	Database	SQL Server	Ileri Seviye SQL Server T-SQL	9
6	Database	SQL Server	SQL Server Veritabani Yönetimi	9
7	Database	SQL Server	SQL Server 2012 Yenilikleri	7
8	Database	Oracle	Ileri Seviye Oracle PL/SQL	9
9	Database	Oracle	Oracle PL/SQL	9
10	Database	Oracle	Oracle Veritabani Yönetimi	9
11	Database	Oracle	Oracle Is Uygulamalari	9

ROLLUP deyimini kullanarak sorgulama yapalım.

```
SELECT Category, SubCategory, BookName,
SUM ([Population]) AS [Population]
FROM tbPopulation
GROUP BY Category,SubCategory,BookName WITH ROLLUP;
```

	Category	SubCategory	BookName	Population
1	Database	Oracle	Ileri Seviye Oracle PL/SQL	9
2	Database	Oracle	Oracle Is Uygulamalari	9
3	Database	Oracle	Oracle PL/SQL	9
4	Database	Oracle	Oracle Veritabani Yönetimi	9
5	Database	Oracle	NULL	36
6	Database	SQL Server	Ileri Seviye SQL Server T-SQL	9
7	Database	SQL Server	SQL Server 2012 Yenilikleri	7
8	Database	SQL Server	SQL Server Veritabani Yönetimi	9
9	Database	SQL Server	NULL	25
10	Database	NULL	NULL	61
11	Software	Android	Ileri Seviye Android Programlama	9
12	Software	Android	NULL	9
13	Software	Java	Ileri Seviye Java Programlama	8
14	Software	Java	NULL	8
15	Software	PHP	PHP 6	6
16	Software	PHP	PHP 6 ile E-Ticaret Uygulamas...	6
17	Software	PHP	NULL	12
18	Software	NULL	NULL	29
19	NULL	NULL	NULL	90

GROUPING İLE ÖZETLERİ DÜZENLEMEK

Bir satır, **ROLLUP** ya da **CUBE** deyimi tarafından türetilmiş ise 1, türetilmemiş ise 0 değeri döndürür.

```
SELECT SalesQuota, SUM(SalesYTD) 'TotalSalesYTD',
  GROUPING(SalesQuota) AS 'Grouping'
FROM Sales.SalesPerson
GROUP BY SalesQuota WITH ROLLUP;
```

	SalesQuota	TotalSalesYTD	Grouping
1	NULL	1252127,9471	0
2	250000,00	27370537,97	0
3	300000,00	7654925,9863	0
4	NULL	36277591,9034	1