

VERİ BÜTÜNLÜĞÜNÜ KAVRAMAK

4

VERİ BÜTÜNLÜĞÜNÜ KAVRAMAK

Tablo işlemleri veri yönetiminin çekirdeğini oluşturur. Tabloların görevi; sadece verilerin depolanmasını sağlamak değil, aynı zamanda sütunlara eklenmek istenen verinin kontrolünü sağlamak, veriler üzerinde kıyaslamalar yapmak, sütunların ilişkili olduğu diğer sütunlar ile arasındaki ilişkinin koparılmasını sağlamaktır. Bir başka sütun ile ilişkili olan bir sütundaki verinin silinmesi ya da değiştirilmesi, diğer sütundaki bağlantılı olduğu verinin ilişkiden koparılması, yani bağlantısız veri haline gelmesini sağlar. Veri tutarlılığını sağlamak için gerekli özellikleri bu bölümde inceleyeceğiz.

TANIMLAMALI VERİ BÜTÜNLÜĞÜ

Nesnelerin kendi tanımları ile elde edilen veri bütünlüğüne denir. Tanımlamalı veri bütünlüğü nesneleri, **Rule**'lar, **Default**'lar ve **Constraint**'ler olmak üzere üç çeşittir.

Bu nesne tiplerinden en iyi performansa sahip olanı ve en etkili **Constraint**'lerdir. Bölümün ilerleyen konularında bu üç nesne tipine de detaylıca değineceğiz.

PROSEDÜREL VERİ BÜTÜNLÜĞÜ

Teknik olarak, tanımlamalı veri bütünlüğünden bir üst seviye veri bütünlüğü kavramıdır. Tanımlamalı veri bütünlüğü SQL Server tarafından oluşturulan

özellik ve kontrollerden oluştuğu için daha alt seviye ve çoğu zaman performanslı olan yöntemdir. Prosedürel veri bütünlüğünün amacı, tanımlamalı veri bütünlüğü özelliklerinin yetersiz kalabileceği bazı durumlarda, programcıya kendi veri bütünlüğü modelini oluşturabilmesi için esneklik sağlamaktır.

Bu yöntem için **Trigger** ve **Stored Procedure** gibi programlama nesneleri kullanılır.

CONSTRAINT TIPLERİ

Constraint'ler kategorisel anlamda üç gruba ayrılır.

- **Domain Constraint**

Belirli sütun ya da sütunlar kümesinin belirli kriterlere uygun olmasını sağlar. Sütun bazında veri kontrolü için kullanılır. Örneğin; doğum tarihi değeri girilmesi istenen bir tablo tasarımında, genellikle 1900 yılı öncesinde bir doğum tarihi seçilmesine izin verilmez. Bu mantıksal bir kontroldür. Bu işlemi veritabanında gerçekleştirebilmek için de Domain Constraint'ler kullanılır.

Bunlar; **Rule** ve **Default** nesneleri, **Check** ve **Default** constraint'lerdir. Belirtilen Rule ve Default nesneleri, Check ve Default constraint'ler ile aynı işi yaparlar.

- **Entity Constraint**

Satır bazlı çalışan constraint'lerdir. Bir satırdaki sütunun **unique** (*benzersiz*) değerlere sahip olmasını sağlamak buna bir örnek olabilir.

Bunlar; **Primary Key** ve **Unique** constraint'lerdir.

- **Referential Integrity Constraint**

Bir sütundaki değer, aynı tablo ya da farklı bir tablodaki farklı bir sütun ile ilişkilendirmek için kullanılır.

Makalelerin tutulduğu **Makaleler** tablosunda, makalenin kategori ID değerini tutan **KategoriID** sütunu ile **Kategoriler** tablosundaki **KategoriID** sütununun birbiriyle ilişkili olma durumunu gerçekleştirir. Kategoriler tablosunda 3 no'lu kategori yok ise, **Makaleler** tablosundaki **KategoriID** sütununda 3 değerinin bulunması ne kadar mantıklı olabilir? Bu ilişkiyi veri bütünlüğünü sağlamak için **Referential Integrity Constraint** kullanılır. Bu constraint'lere örnek olarak **Foreign Key Constraint** verilebilir.

CONSTRAINT İSİMLENDİRMESİ

Constraint oluşturmanın birkaç farklı yöntemi vardır. Management Studio ile, bir script ile oluşturmak ya da programcının kendisi yazarak oluşturabilir. Bu farklı durumlarda SQL Server'ın constraint isimlendirme politikası farklı olabilir.

Örneğin, bir script ile constraint ismi belirtmeden otomatik oluşturmayı seçtiğimizde SQL Server, bir constraint'i şu şekilde isimlendirir.

```
PK__Accounts__349DA5862A269D65
```

Yukarıdaki isimlendirmede **PK** tanımı **Primary Key**, **Accounts** ise constraint tanımlanan tablonun adıdır. Sonrasındaki Unique değer ise, SQL Server tarafından oluşturulan bir isimlendirme ekidir.

SQL Server, **Primary Key** için **PK**, **Unique Constraint** için **UQ**, **Check Constraint** için **CK** isim ön ekini kullanır. Programcı olarak biz de, kendimiz isimlendirme oluştururken bu kurallara uyabiliriz.

Bu constraint **Management Studio** ile oluşturulsaydı, ismi **PK_Accounts** olacaktı.

Programcı olarak yukarıdaki gibi, uzun ve karışık bir isimlendirme modelini kullanmanızı önermiyorum.

Bu tür karmaşık isimlendirmeler anlaşılması zor olduğu gibi, hatırlanması ve hangi amaç ile oluşturulduğunun bilinmesi de zordur. Bu nedenle, SQL Server'ın sizin amacınızı anlayarak buna göre isimlendirme yapmasını beklemeyin. Kendi constraint isimlendirme modelinizi kullanın.

İlerleyen konularda bolca göreceğiniz gibi constraint isimlendirirken anlamlı, kısa ve anlaşılır isimler belirtilmelidir.

Örneğin, e-mail bilgilerini tutan bir sütunda e-mail formatını belirleyen bir Check Constraint oluşturmak için şu isimlendirmeler kullanılabilir.

```
CK_Accounts_Email  
CK_Email  
CKEmail
```

İsimlendirmeler kısa, anlaşılır ve tutarlı olmalıdır. Yani, isimlendirme için bir model belirlediyseniz bu modeli tüm veritabanı geliştirmesi süresince kullanmalısınız.

SÜTUN SEVİYELİ VERİ BÜTÜNLÜĞÜ

Sütun seviyeli constraint'ler sütunlara girilecek veriler için denetleme gerçekleştirmeye yarayan veritabanı nesneleridir. Bir sütunun boş geçilememesi (**NOT NULL**), ya da doğum tarihi değeri girilirken belirli bir formatta ve tarih aralığında (Örn; 1900'den şimdiki tarihe kadar) veri girişi yapılmasını sağlayan ve farklı veri girişi isteklerini engelleyen yapılardır.

PRIMARY KEY CONSTRAINT OLUŞTURMAK

Primary key'ler, her satır için kullanılan **unique** (*benzersiz*) tanımlayıcılardır. Unique değerler içermek zorundadırlar. Bir tabloda sadece bir tane **Primary Key Constraint** tanımlanabilir. Genel olarak tüm tablolarda kullanılsa da, zorunlu değildir. Bazen unique bir değer kullanma ihtiyacı olmayan tablolar oluşturulması gerekebilir.

Primary key, bildirildiği sütunlarda unique özelliğinin sağlanmasını garanti eder ve asla **NULL** değer içeremez.

Primary Key, **SSMS** ya da **T-SQL** ile oluşturulabilir. Mantıksal olarak da, ya tablo oluşturulurken (**CREATE**) ya da tablo değiştirilirken (**ALTER**) oluşturulabilir.

TABLO OLUŞTURMA SIRASINDA PRIMARY KEY OLUŞTURMAK

Yeni bir tablo oluştururken, aşağıdaki gibi bir Primary Key Constraint oluşturulabilir.

```
CREATE TABLE Urunler(
    UrunID          INT,
    UrunAd          VARCHAR(200),
    UrunFiyat       MONEY,
    CONSTRAINT PKC_UrunID PRIMARY KEY(UrunID)
);
```

ya da

```
CREATE TABLE Urunler(
    UrunID INT IDENTITY NOT NULL PRIMARY KEY,
    UrunAd VARCHAR(200),
    UrunFiyat MONEY
);
```

Tablo oluşturulduktan sonra, şu şekilde constraint'ler izlenebilir.

```
sp_helpconstraint 'Urunler';
```

Object Name							
1	Urunler						
	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	PKC_UrunID	(n/a)	(n/a)	(n/a)	(n/a)	UrunID

MEVCUT BİR TABLODA PRIMARY KEY OLUŞTURMAK

Bir tabloya, oluşturulduktan sonra Primary Key eklemek için aşağıdaki söz dizimi kullanılır.

```
ALTER TABLE tablo_ismi
ADD CONSTRAINT constraint_ismi PRIMARY KEY(sutun_ismi)
[CLUSTERED|NONCLUSTERED]
```

Daha önce oluşturulan bir tabloya, sonradan bir Primary Key ekleme senaryosu **ALTER** ile şu şekilde gerçekleştirilir.

Kullanıcı bilgilerini tutacak test tablomuzu oluşturalım.

```
CREATE TABLE Kullanicilar(
    KullanciID          INT PRIMARY KEY NOT NULL,
    Ad                  VARCHAR(50),
    Soyad               VARCHAR(50),
    KullanciAd          VARCHAR(20)
);
```

Artık oluşturduğumuz tabloya **ALTER** ile bir **Primary Key** ekleyebiliriz.

```
ALTER TABLE Kullanicilar
ADD CONSTRAINT PKC_KullanciID PRIMARY KEY(KullanciID);
```

Bu işlemle, **KullanciID** sütunu için bir Primary Key Constraint oluşturuldu. **ALTER** işlemiyle Primary Key eklerken, Primary Key olması istenen sütunun **NOT NULL** olmasını garanti etmelisiniz. Sütun daha önceden var olduğu için, **NULL** geçilebilir olarak oluşturulmuş olabilir. Bu nedenle, **ALTER** ifadesi ile **NOT NULL** belirtimi yapılmalıdır. Aksi halde, **ALTER** işlemi sırasında hata alınır.

UNIQUE KEY CONSTRAINT OLUŞTURMAK

Unique Key Constraint tanımlı bir sütun **NULL** değer içerebilir. Ancak eğer bir değer girilecek ise bu değer benzersiz (*unique*) olmalıdır. Bir tabloda birden fazla Unique Key Constraint tanımlanabilir.

Unique Key Constraint ile işaretlenen bir sütun **NULL** değer içerebilir. Ancak sütunda sadece bir kez **NULL** kullanılabilir. Birden fazla **NULL** değer tanımlaması yapılmasını engellemek için **CHECK** Constraint kullanılabilir.

Unique Key Constraint tanımlandıktan sonra, isimlendirilen her sütundaki her değer unique olmak zorundadır. Kayıtlarda var olan bir değer ekleme ya da güncellenmek istenirse SQL Server hata vererek işlemi reddedecektir.

TABLO OLUŞTURMA SIRASINDA UNIQUE KEY CONSTRAINT OLUŞTURMAK

Tablo oluşturma sırasında iki şekilde Unique Key Constraint oluşturulabilir.

Personeller adında bir tablo oluştururken **KullaniciAd** sütununun boş geçilmemesini ve aynı değerlerin kullanılamamasını garanti edecek şekilde bir Unique Key Constraint oluşturalım.

```
CREATE TABLE Personeller
(
    PersonelID          INT PRIMARY KEY NOT NULL,
    Ad                  VARCHAR(255) NOT NULL,
    Soyad               VARCHAR(255) NOT NULL,
    KullaniciAd         VARCHAR(10) NOT NULL UNIQUE,
    Email               VARCHAR(50),
    Adres               VARCHAR(255),
    Sehir               VARCHAR(255),
);
```

Aynı işlem şu şekilde de gerçekleştirilebilir.

```
CREATE TABLE Personeller
(
    PersonelID          INT PRIMARY KEY NOT NULL,
    Ad                  VARCHAR(255) NOT NULL,
    Soyad               VARCHAR(255) NOT NULL,
```

```

KullaniciAd    VARCHAR(10) NOT NULL,
Email          VARCHAR(50),
Adres          VARCHAR(255),
Sehir          VARCHAR(255),
UNIQUE (KullaniciAd)
);

```

Tablo oluşturulduktan sonra, şu şekilde constraint'ler izlenebilir.

```
sp_helpconstraint 'Personeller';
```

Object Name							
1	Personeller						
	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	PK_Personel__0F0C5751058E741F	(n/a)	(n/a)	(n/a)	(n/a)	PersonelID
2	UNIQUE (non-clustered)	UQ_Personel__E0103670690944F2	(n/a)	(n/a)	(n/a)	(n/a)	KullaniciAd

MEVCUT BİR TABLODA UNIQUE KEY OLUŞTURMAK

Önceden var olan bir tabloda Unique Key Constraint oluşturmak kolaydır.

Oluşturduğumuz **Personeller** tablosunda **Email** sütununun da **Unique** olmasını istiyoruz.

```

ALTER TABLE Personeller
ADD CONSTRAINT UQ_PersonelEmail
UNIQUE (Email)

```

Constraint oluşturulurken SQL Server tarafından otomatik olarak bir isimlendirme kuralı kullanılır. **Primary Key**'de **PK**, **Unique**'de ise **UQ** isimlendirmesidir.

Yönetilebilirlik ve sistem üzerinde, bazı kurallar oluşturabilmeniz, hangi constraint'i tabloya sonradan dahil ettiğinizi hatırlayabilmeniz için, **ALTER** ifadesi ile sonradan eklediğiniz constraint'lerde **AK** isimlendirmesini kullanabilirsiniz. Bu sayede, constraint'leri görüntülediğinizde hangi constraint'in sonradan eklendiğini fark etmeniz kolay olacaktır.

DEFAULT CONSTRAINT

DEFAULT constraint, SQL Server'da varsayılan değer yerine geçecek bir değer tanımlama işlemi için kullanılır.

Örneğin; kullanıcılar tablosunda kullanıcının kayıt olduğu zaman, bilgisi istemci tarafından parametre olarak bildirilmek zorunda değildir. SQL Server tarafında, kullanıcılar tablosundaki kayıt tarih sütununa bir **DEFAULT** tanımlayarak, o anki sistem zaman bilgisinin kayıt sırasında ilgili sütuna otomatik olarak eklenmesi sağlanabilir.

SQL Server'da **NULL** alanlar oldukça can sıkıcıdır. Birçok hataya sebep olabilir ve ek yönetim gereksinimine ihtiyaç duyarlar. Bu durum **DEFAULT** constraint ile çözülebilir. Ürünlerin tutulduğu bir tabloda, ürün fiyat alanında eğer değer girilmemiş ürünler var ise, bu ürünlere varsayılan olarak 0 (sıfır) değeri atanması sağlanabilir. Bu çözüm ile en azından programlama tarafında bir değer olarak görülebilecek 0 değerine sahip olunacaktır.

- **DEFAULT** constraint'ler, sadece **INSERT** cümlelerinde kullanılabilir. **UPDATE** ve **DELETE** ifadelerinde yok sayılır.
- **INSERT** ifadesinde, **DEFAULT** kullanılan sütun için farklı bir değer belirtilmişse, **DEFAULT** yok sayılır ve belirtilen değer sütuna eklenir.

TABLO OLUŞTURURKEN DEFAULT CONSTRAINT TANIMLAMA

CREATE ifadesi ile birlikte, tablo oluşturulurken bir **DEFAULT** constraint oluşturmak için, **Personeller** tablosunu yeni sütunlarıyla tekrar oluşturalım.

Daha önce oluşturulan **Personeller** tablosunu silerek yeniden oluşturalım.

```
DROP TABLE Personeller
```

Personeller tablosunu farklı özelliklerle yeniden oluşturalım.

```
CREATE TABLE Personeller
(
    PersonelID INT          PRIMARY KEY NOT NULL,
    KullaniciAd  VARCHAR(20) NOT NULL,
    Email        VARCHAR(50),
    Sehir        VARCHAR(50),
    KayitTarih   SMALLDATETIME NOT NULL DEFAULT GETDATE()
);
```

Personeller tablosuna veri eklemek için **INSERT** ifadesi kullanılırken, artık **KayıtTarih** sütununa herhangi bir değer girme zorunluluğu yoktur. SQL Server, **Personeller** tablosuna bir veri ekleme sorgusu fark ettiğinde, sistem tarih ve zaman bilgisini otomatik olarak bu sütuna ekleyecektir.

Bir veri ekleyerek bu işlemi deneyelim.

```
INSERT INTO Personeller(PersonelID, KullaniciAd, Email, Sehir)
VALUES (1, 'SamiAyyildiz', 'samil.ayyildiz@abc.com', 'İstanbul');
```

Eklenen veriyi listeleyelim.

```
SELECT * FROM Personeller;
```

	PersonelID	KullaniciAd	Email	Sehir	KayıtTarih
1	1	SamiAyyildiz	samil.ayyildiz@abc.com	İstanbul	2013-01-30 13:38:00

Görüldüğü gibi, **INSERT** ifadesinde kayıt tarih sütununu seçmememize ve bir değer belirtmememize rağmen, ilgili sütuna **GETDATE()** fonksiyonu ile zaman bilgisi eklendi.

Oluşturulan yeni **DEFAULT** constraint'ı görebilirsiniz.

```
sp_helpconstraint 'Personeller';
```

Object Name							
1	Personeller						
	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	DEFAULT on column KayıtTarih	DF__Personell__Kayit__2B5F6B28	(n/a)	(n/a)	(n/a)	(n/a)	(getdate())
2	PRIMARY KEY (clustered)	PK__Personel__0F0C5751BAA38D7D	(n/a)	(n/a)	(n/a)	(n/a)	PersonelID

VAR OLAN TABLOYA DEFAULT CONSTRAINT EKLEMEK

Personeller tablosunun var olduğunu ve sonradan bir **DEFAULT** constraint eklemek istediğimizi düşünürsek, işlem şu şekilde gerçekleştirilir.

```
ALTER TABLE Personeller
ADD CONSTRAINT DC_KayıtTarih DEFAULT GETDATE() FOR KayıtTarih
```

Bu işlem, metinsel değer taşıyan bir sütun üzerinde de yapılabilirdi.

```
ALTER TABLE Personeller
ADD CONSTRAINT DC_Sehir DEFAULT 'Tanımsız' FOR Sehir
```

DEFAULT NESNESİ

Default nesnesi, **Default Constraint** ile aynı işleve sahiptir. SQL Server tarafından, geriye doğru uyumluluk için desteklenen bu nesnenin farkı, ayrı bir nesne olarak derlenmesidir. Bir tablonun bir alanı için bir **Default** nesnesi tanımlanabilir.

Default nesnesi geriye doğru uyumluluk için desteklenen bir nesne olması, sonraki SQL Server sürümlerinde desteklenmeyeceği anlamına gelir. **Default** nesnesi kullanmak yerine, **Default Constraint** kullanmanız, SQL Server uyumluluğu açısından daha iyi olacaktır.

Söz dizimi ve kullanımı aşağıdaki gibidir:

```
CREATE DEFAULT default_ismi AS [default_deger | default_ifade]
```

Default nesnesi oluşturulduktan sonra, **sp_bindefault** sistem prosedürü kullanılarak sütun ilişkilendirmesi yapılmalıdır.

```
sp_bindefault default_ismi, 'tablo.sutun_ismi'
```

Default nesnesini silmek için ise **DROP** ifadesi kullanılır.

```
DROP DEFAULT default_ismi
```

CHECK CONSTRAINT

Bir sütuna eklenecek verileri belli kıyaslara karşı kontrol etmek için kullanılır. Bir sütun için birden fazla **Check Constraint** tanımlanabilir.

Örneğin, telefon numaralarının belirli standart karakter sayıları vardır. 11 karakterlik bir telefon numarası için 6 karakter girildiyse, herhangi bir kayıt işlemine gerek yoktur. Çünkü girilecek veri mantıksal olarak hatalıdır. Ya da bir domain adresi istenirken, *www.* ile başlayan ilk kısmı dahil edilmek istenirse en az 5 karakterden oluşmalıdır. Çünkü domain isimleri iki karakterden az karakter içeremez. Bir başka örnek ise, şifre alanları olabilir. Kullanıcıdan bir

şifre belirlenmesi isteniyorsa, genel olarak en az 4 karakterli olması istenir. Hatta yüksek güvenlik için 10 karakterlik bir alt sınır da belirlenebilir.

Şimdi, bir şifre sütunu üzerinde karakter kontrolü gerçekleştirelim.

Check Constraint kontrolü için **Kullaniciilar** isimli yeni bir tablo oluşturalım.

```
CREATE TABLE Kullaniciilar
(
    KullaniciID          INT PRIMARY KEY NOT NULL,
    KullaniciAd          VARCHAR(20) NOT NULL,
    Sifre                VARCHAR(15) NOT NULL,
    Email                VARCHAR(40) NOT NULL,
    Telefon              VARCHAR(11) NOT NULL
);
```

Tablodaki tüm alanlar kritik öneme sahip olduğu için **NULL**, yani boş geçilemez olarak belirledik. Şimdi de constraint'i oluşturalım.

```
ALTER TABLE Kullaniciilar
ADD CONSTRAINT CHK_SifreUzunluk CHECK(LEN(Sifre)
>= 5 AND LEN(Sifre) <= 15)
```

Tablo, key ve nesneler arası ilişkilerin tasarımı çok önemli ve iyi hesaplanması gereken bir konudur. Kullanıcılar tablosunda **sifre** sütununu, maksimum 15 karakter alabilecek şekilde oluşturduk. Constraint tanımlarken ise, en az 5 karakter ve en fazla 15 karakterlik bir değer girilmesini istediğimizi belirttik ve bunu kontrol ettik. Tablodaki **sifre** sütununda 15 karakterlik bir sınırlama varken constraint'de bu 15 karakterlik üst sınırlamayı oluşturmazsak, farklı sorunlara yol açabilir. Bu nedenle constraint, 15 karakterden fazla değer girişi yapılamaz şekilde oluşturuldu. 5 karakterden az ya da 15 karakterden fazla değer girişi yapılırsa hata üretilecektir.

Geliştirilen uygulama her yönüyle hesaplanmalıdır. Kullanıcının en az kaç karakter girmesi istendiği, en fazla kaç karakter girmesini istendiği, bunları daha önceden belirlemeli ve tablo mimarisi buna göre tasarlanmalıdır.

Genellikle üyelik gerektiren ve web uygulamalarında çok kullanılan mail adresinin doğruluğunu test eden yazılımlar vardır. Bu işlemi veritabanı tarafında gerçekleştirmek mümkündür ve gereklidir.

Girilen mail adresi değeri, içerisinde @ işareti için kontrol yapacak ve bu işaret yok ise, kaydetme isteğini reddedecek bir **Check Constraint** oluşturalım.

```
ALTER TABLE Kullanicilar
ADD CONSTRAINT CHK_Email CHECK (CHARINDEX('@', Email) > 0 OR Email IS NULL)
```

Şifre ve e-mail constraint'lerini test etmek için bir veri ekleme işlemi gerçekleştirelim.

```
INSERT INTO Kullanicilar
VALUES (1, 'cihan', 'sifre', 'cihan.ozhan@hotmail.com', '05551112233');
```

Bu sorgu başarılı bir şekilde veri ekleme işlemi gerçekleştirecektir. Çünkü 'sifre' değeri en az 5 karakterlik şifre olma özelliğini taşıyor. Aynı zamanda e-mail adresindeki @ işareti de **Email** sütunu üzerindeki constraint'in isteğini karşıladığı için herhangi bir hata vermeyecektir.

Ancak, şifre sütunundaki değer 4 karakterli olursa ya da e-mail adresindeki karakterler arasında @ işareti olmazsa, veri ekleme işlemi gerçekleşmeyecek, hata verecektir.

Aşağıdaki kullanımların ikisi de, oluşturduğumuz Check Constraint'ler açısından hatalıdır.

Şifre kontrolünden dolayı hata verecektir.

```
INSERT INTO Kullanicilar
VALUES (1, 'cihan', 'sifre1', 'cihan.ozhan@hotmail.com');
```

E-mail kontrolünden dolayı hata verecektir.

```
INSERT INTO Kullanicilar
VALUES (1, 'cihan', 'sifre', 'cihan.ozhan-hotmail.com');
```

Telefon kontrolünden dolayı hata verecektir.

```
INSERT INTO Kullanicilar
VALUES (1, 'cihan', 'sifre', 'cihan.ozhan@hotmail.com', '5551112233');
```

Son olarak, sık kullanılan özelliklerden bir diğeri olan, telefon bilgisi içeren bir sütun için Check Constraint oluşturalım.

İstediğimiz telefon formatı şu: **XXXXXXXXXX**

```
ALTER TABLE Kullanicilar
ADD CONSTRAINT CHK_Telefon CHECK(
Telefon IS NULL OR(
Telefon LIKE '[0][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' )
AND LEN(Telefon) = 11)
```

Telefonun ilk karakterinin 0 (*sıfır*) olmasını ve kalan karakterlerinde doğal olarak nümerik tekli sayılar arasında (0-9) olmasını şart koştuk. Bu yapı ve karakter sayısına uymayan istekler reddedilecektir.

Bu formata uyum sağlayacak örnek telefon: 05420425262

Farklı bir format oluşturulmak istenebilir. Örneğin; 0542-042-52-62

Bu şekilde bir formatlama da aşağıdaki gibi yapılabilir:

```
[0][0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]
```

İlk 4 karakterden sonra - özel karakterini, sonraki 3 karakterden sonra ve sonraki 2 karakterden sonra aynı karakteri yerleştirdik.

Burada dikkat edilmesi gereken nokta şu ki; ayıraç olarak eklenen tüm karakterler telefon numarası karakter sayısını artırmaktadır. Ayıraç (-) kullanıldıktan sonraki sorgu şu şekilde olmalıdır.

```
ALTER TABLE Kullanicilar
ADD CONSTRAINT CHK_Telefon CHECK(
Telefon IS NULL OR(
Telefon LIKE '[0][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' )
AND LEN(Telefon) = 14)
```

3 özel karakteri de ekleyerek toplam 14 karakterlik bir veri girişi yapılması gerekir. Bu durumda tabi ki tablodaki ilişkili sütunun karakter sınırı da 11 değil, 14 olarak tanımlanmalıdır.

RULE

Default Constraint yerine **Default** nesnesi kullanılabildiği gibi, **Check Constraint**'in gerçekleştirdiği işlemi, Rule nesnesi de gerçekleştirebilir.

Söz Dizimi:

```
CREATE RULE rule_ismi
AS ifadeler
```

Tanımlanan rule'ü ilgili sütuna ilişkilendirmek için şu yapı kullanılır:

```
sp_bindrule rule_ismi, tablo.sutun_ismi
```

Tanımlanan rule'ü silmek için ise şu yapı kullanılır:

```
DROP RULE rule_ismi
```

TABLO SEVİYELİ VERİ BÜTÜNLÜĞÜ

Tablo seviyeli varlık bütünlüğünü sağlamak, sütunlar arasında ve tablolar arasında birbiri ile uyumlu olmasını sağlayacağız.

SÜTUNLAR ARASI CHECK CONSTRAINT

Bazen iki sütunun değerini kontrol ederek bir işlem yapmak gerekebilir. Örneğin; bir kulüp üyelerinin tutulduğu bir yönetim yazılımı hazırlanıyor. Bu yazılımda kullanıcıların üyelik tarihi ve üyeliğini iptal ederek üyelikten çıkanlar varsa çıkış tarihini tutsun.

Mantıksal olarak, bir üyelik tarihinden önceki bir tarih, üyenin çıkış tarihi olamaz. Ancak bunu yazılımsal olarak kontrol etmezsek, mantık olarak gerçekleşmeyecek bir şey teknik olarak gerçekleştirilebilir. Buna da yazılımda **mantık hatası** denir.

Üyelerin bulunduğu tabloyu oluştururken aynı zamanda Constraint'de oluşturalım.

```
CREATE TABLE Uyeler
(
    UyelerID INT PRIMARY KEY NOT NULL,
```

```

UyelikAd          VARCHAR(20) NOT NULL,
Sifre             VARCHAR(10) NOT NULL,
Email            VARCHAR(30),
Telefon           VARCHAR(11),
GirisTarih        DATETIME,
CikisTarih        DATETIME NULL,
CONSTRAINT       CHK_CalismaTarih CHECK(
CikisTarih        IS NULL OR CikisTarih >= GirisTarih)
);

```

Oluşturduğumuz tabloya iki adet kayıt girme denemesi gerçekleştireceğiz.

Başarıyla çalışması beklenen, doğru verilere sahip bir veri ekleme yapalım.

```

INSERT INTO Uyeler
VALUES(1,'cihan','sifre','cihan.ozhan@hotmail.com','05310806080',
      '2011-01-15','2011-11-25');

```

Bu veri ekleme işleminin başarılı olmasının sebebi **GirisTarih** değerinin **CikisTarih** değerinden küçük olmasıdır. Şimdi, bu durumun tam tersi olarak **CikisTarih** değeri **GirisTarih** değerinden küçük bir kayıt girişi yapmaya çalışalım.

```

INSERT INTO Uyeler
VALUES(1,'cihan','sifre','cihan.ozhan@hotmail.com','05310806080',
      '2011-01-15','2010-11-25');

```

Sorgunun hata vermesinin sebebi, **CikisTarih** sütun değerine 2010 tarihli bir değer atamak istememizdir. Mantık olarak 2011 tarihinde kulübe kaydolun biri 2012 tarihinde çıkış yapamaz.

FOREIGN KEY CONSTRAINT

Foreign Key, tablolar arası veri doğruluğunun sağlanması ve tablolar arası ilişkilendirmelerin gösterilmesinin bir yoludur. Tabloya bir Foreign Key eklendiğinde, Foreign Key tanımlanan tablo (referans alan tablo) ile Foreign Key'in referans olarak aldığı tablo (referans alınan tablo) arasında bir bağlantı oluşturulur.

Foreign Key eklenmesinden sonra, referans alan tabloya eklenen her kayıt, referans alınan tabloda referans alınan sütunlardaki kayıt ile eşleşmek zorundadır. Yani, aralarında kırılmaz bir zincir var da denebilir.

Bir örnek Foreign Key uygulaması için iki tablo oluşturalım.

Bağlantı kurulan **Kategoriler** tablosunu oluşturalım.

```
CREATE TABLE Kategoriler
(
    KategoriID          INT IDENTITY PRIMARY KEY,
    KategoriAd          VARCHAR(20)
);
```

Bağlantı kuracak Makaleler tablosunu oluşturalım.

```
CREATE TABLE Makaleler
(
    MakaleID            INT IDENTITY PRIMARY KEY,
    Baslik              VARCHAR(100),
    Icerik              VARCHAR(MAX),
    KategoriID          INT FOREIGN KEY REFERENCES Kategoriler(KategoriID),
    EklenmeTarih        DATETIME
);
```

Referans alınacak tablodaki gerçek sütunda **PRIMARY KEY** ya da **UNIQUE** Constraint tanımlanmış olması gerekir. Aksi halde hata oluşacaktır.

VAR OLAN TABLOYA

FOREIGN KEY CONSTRAINT EKLEMEK

Daha önce oluşturulmuş bir tabloya Foreign Key Constraint eklemek diğer constraint değiştirme örnekleriyle aynıdır.

```
ALTER TABLE Makaleler
ADD CONSTRAINT FK_MakaleKategoriler
FOREIGN KEY(KategoriID) REFERENCES Kategoriler(KategoriID)
```

CONSTRAINT'LERİ DEVRE DIŞI BIRAKMAK

Bazen constraint ile gerçekleştirilen kontroller, yani constraint'ler bir süreliğine ya da tamamen kaldırmak istenebilir. Bu SQL Server'da mümkündür.

Primary Key ve **Unique** constraint'ler hariç, diğer constraint'ler devre dışı bırakılabilir.

BOZUK VERİYİ İHMAL ETMEK

Constraint'lerin kullanımı bazı felaket senaryolarının önüne geçecek kadar önemli olabilir. Genel olarak bir tablo oluşturulurken constraint oluşturuluyor olsa da, geriye dönük desteklenen veritabanlarının modernize edilmesi, algoritma değişikliği gibi durumlar söz konusu olabilir.

Kullanıcıların bilgilerini tutan bir tabloda, telefon bilgilerinin tutulduğu bir sütun olması senaryosunu inceleyelim.

Daha önce telefon sütunu vardı ancak üzerinde herhangi bir kontrol işlemi gerçekleştirilmiyordu. Yani, bir constraint yoktu. Sonradan bir Unique Constraint oluşturarak bir telefon formatı belirledik. Artık herkes bu formatta veri girişi yapmak zorunda olacaktır. Peki, önceki var olan kayıtların uyumluluğu ne olacak?

Bu senaryoda, normal constraint kullanımında eski telefon kayıtlarından dolayı **ALTER TABLE** işlemi hata verecektir.

Bu işlemin hata vermemesi için **WITH NO CHECK** özelliği kullanılmalıdır.

```
ALTER TABLE Kullanicilar
WITH NO CHECK
ADD CONSTRAINT CHK_Telefon CHECK(
Telefon IS NULL OR(
Telefon LIKE '[0][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' )
AND LEN(Telefon) = 14)
```

WITH NO CHECK ile önceki formatlanmamış telefon verilerinden dolayı, constraint engellemesi ortadan kalkacaktır. Bu işlemten sonra gerçekleştirilecek veri eklemelerinde, artık bu telefon formatına uyulmak zorundadır.

CONSTRAINT'İ GEÇİCİ OLARAK DEVRE DIŞI BIRAKMAK

SQL Server'ın geriye dönük uyumluluk özelliklerinden birisi de **NOCHECK** komutudur. Tablonuzda constraint ile kontrol yapıyorsunuz, ancak eski ve kontrol kurallarınıza uymayan kayıtların bulunduğu veri kaynaklarından bu tabloya veri aktarma işlemi gerçekleştirmek istiyorsunuz. Bu durumda, aktarma işlemi yapmak hataya sebebiyet verir ve aktarma gerçekleşmez.

Ancak, SQL Server'ın bir süreliğine bu duruma göz yummasını sağlayabiliriz.

NOCHECK özelliği ile constraint'lerin kontrollerinden kurtulmak mümkündür.

```
ALTER TABLE Kullanicilar  
NOCHECK  
CONSTRAINT CHK_Telefon
```

Yukarıdaki sorgu ile artık farklı formatlardaki telefon numaralarının bulunduğu kayıtları, yeni ve constraint ile kontrol edilen tabloya aktarılabilir.

