

## İÇİNDEKİLER

### 1 VERİTABANI KAVRAMINA GİRİŞ

Veritabanı Nedir?	1
SQL, T-SQL ve Veritabanı Programlama	2
İlişkisel Veritabanı Yönetim Sistemi	3
SQL Server Nedir?	4
Neden SQL Server?	4
SQL Server 2012 Sürümü ve Özellikleri	6
SQL Server 2012 Sürümüleri	6
SQL Server 2012 Express Edition	7
SQL Server 2012 Developer Edition	7
SQL Server 2012 Web Edition	7
SQL Server 2012 Business Intelligence Edition	7
SQL Server 2012 Standard Edition	7
SQL Server 2012 Enterprise Edition	8
SQL Server 2012 Compact Edition	8
SQL Server Lisanslama	8
SQL Server Kurulumu & Kurulumu Kaldırma	9
SQL Server 2012 Kurulum Gereksinimleri	9
SQL Server 2012 Kurulum & Kurulumu Kaldırma İşlemleri	10
SQL Server Araçları	11
Books Online	11
SQL Server Configuration Manager	11
Servis Yönetimi	12
Ağ Yapılandırması	13
Protokoller	14
TCP/IP	15
Named Pipes	15
Shared Memory	15

VIA	15
SQL Server Management Studio	16
SQL Server Profiler	16
SQL Server Integration Services (SSIS)	16
SQL Server Reporting Services	17
SQL Server Business Intelligence	17
Development Studio	17
Bulk Copy Program (BCP)	17
sqlcmd	18
Özet	18
SQL Server Veritabanı Nesneleri	18
Bu Bölümde Neler Öğreneceğiz?	19
Veritabanı Nesnelerine Genel Bakış	19
Database Nesnesi	19
master	20
model	21
msdb	21
tempdb	21
AdventureWorks	22
AdventureWorksDW	22
Transaction Logs (İşlem Günlükleri)	22
Filegroups (Dosya Grupları)	23
Diagrams (Diyagramlar)	23
Schemas (Şemalar)	23
Tables (Tablolar)	24
Indexes (İndeksler)	24
Constraints (Kısıtlamalar)	25
Views (Görünümler)	25
Stored Procedures (Saklı Yordamlar)	25
Triggers (Tetikleyiciler)	26

User-Defined Functions (Kullanıcı Tanımlı Fonksiyonlar)	26
User-Defined Data Types (Kullanıcı-Tanımlı Veri Tipleri)	27
Defaults	27
Users & Roles	27
Rules	27
Tanımlayıcılar ve İsimlendirme Kuralları	28
İsimlendirme Kuralları	28
<b>2 T-SQL'E GENEL BAKIŞ</b>	<b>31</b>
Transact-SQL Kavramı	31
T-SQL ile İlgili Kurallar	32
Nesne ve Değişken İsimlendirme Kuralları	32
Tanımlayıcı İsimlendirme Notasyonları	33
Notasyon Kavramı	33
Camel Notasyonu	33
Pascal Notasyonu (Deve Notasyonu)	33
Alt Çizgi (Underscore) Notasyonu	33
Büyük Harf (Uppercase) Notasyonu	33
Macar Notasyonu	34
Açıklama Satırları	34
NULL Kavramı	34
T-SQL'de Yığın Kavramı	35
GO Komutu	35
USE Komutu	35
PRINT Komutu	35
Veri Tanımlama Dili	36
SQL Server'da Nesne İsimleri	36
Schema İsmi (Ownership / Sahiplik)	38
Varsayılan Schema: dbo	38
Veritabanı İsmi	39
Server Tarafından İsimlendirme	40

CREATE ile Nesne Oluşturmak	40
CREATE DATABASE ile Veritabanı Oluşturmak	40
ON	41
NAME	41
FILENAME	41
Veri Terimleri Büyüklükleri	42
SIZE	43
MAXSIZE	43
FILEGROWTH	44
LOG ON	44
COLLATE	44
FOR ATTACH	45
DB_CHAINING ON   OFF	45
TRUSTWORTHY	45
T-SQL ile Veritabanı Oluşturmak	46
CREATE TABLE ile Tablo Oluşturmak	49
Tablo ve Sütun İsimleri	49
Veri Tipleri	50
DEFAULT	50
IDENTITY	50
NOT FOR REPLICATION	51
ROWGUIDCOL	51
COLLATE	52
NULL / NOT NULL	52
Sütun Kısıtlamaları	52
Hesaplanmış Sütunlar	53
Tablo Kısıtlamaları	53
ON	53
Tablo Oluşturmak	53
ALTER ile Nesneleri Değiştirmek	54

Veritabanını Değiştirmek	54
Veritabanı Tablosunu Değiştirmek	56
DROP ile Nesne Silmek	57
Bir Veritabanı Tablosunu Silmek	58
Bir Veritabanını Silmek	58
<b>Veri İşleme Dili (DML = Data Manupluation Language)</b>	59
INSERT ile Veri Ekleme	59
SELECT ile Veri Seçmek	60
UPDATE ile Veri Güncellemek	62
DELETE ile Veri Silmek	63
<b>Veri Kontrol Dili (DCL = Data Control Language)</b>	63
GRANT ile Yetki Vermek	64
WITH GRANT OPTION ile Basamaklı Yetkilendirme	65
DENY ile Erişimi Kısıtlamak	66
REVOKE ile Erişim Tanımını Kaldırmak	66
<b>3 VERİLERİ SORGULAMAK</b>	67
Operatör Türleri	67
Aritmetik Operatörleri	68
Atama Operatörü	68
Metin Birleştirme Operatörü	69
SELECT ile Kayıtları Seçmek	69
DISTINCT ile Tekile İndirgemek	70
UNION ve UNION ALL ile Soru Sonuçlarını Birleştirmek	71
WHERE ile Soru Sonuçlarını Filtrelemek	72
Mantıksal Operatörler	72
AND	72
OR	73
Karşılaştırma Operatörleri	75
LIKE	76
Joker Karakterler	77

Belirli Kayıtlar Arasında Sorgulama Yapmak	83
BETWEEN .. AND ..	83
IN ve NOT IN	85
SQL Server'da NULL ve Boşluk Kavramı	88
Metinsel Değerler ile NULL Kullanımı	88
SPACE Fonksiyonu	89
Sorgularda NULL Değer İşlemleri	90
IS NULL Operatörü	90
ISNULL Fonksiyonu	92
COALESCE Fonksiyonu	94
NULIF Fonksiyonu	97
SELECT ile Verileri Sıralamak	97
ORDER BY	98
TOP Operatörü	102
TOP Fonksiyonu	103
Tabloları Birleştirmek	104
Klasik JOIN	105
SQL Server'da JOIN Mimarisi	106
INNER JOIN	108
OUTER JOIN	109
OUTER JOIN Tipleri	109
LEFT OUTER JOIN	109
RIGHT OUTER JOIN	110
FULL OUTER JOIN	111
CROSS JOIN	112
<b>4 VERİ BÜTÜNLÜĞÜNÜ KAVRAMAK</b>	<b>115</b>
Veri Bütünlüğünü Kavramak	115
Tanımlamalı Veri Bütünlüğü	115
Prosedürel Veri Bütünlüğü	115
Constraint Tipleri	116

Constraint İsimlendirmesi	117
Sütun Seviyeli Veri Bütünlüğü	118
Primary Key Constraint Oluşturmak	118
Tablo Oluşturma Sırasında	118
Primary Key Oluşturmak	118
Mevcut Bir Tabloda	119
Primary Key Oluşturmak	119
Unique Key Constraint Oluşturmak	120
Tablo Oluşturma Sırasında	120
Unique Key Constraint Oluşturmak	120
Mevcut Bir Tabloda Unique Key Oluşturmak	121
Default Constraint	122
Tablo Oluştururken	122
DEFAULT Constraint Tanımlama	122
Var Olan Tabloya DEFAULT Constraint Eklemek	123
Default Nesnesi	124
Check Constraint	124
Rule	128
Tablo Seviyeli Veri Bütünlüğü	128
Sütunlar Arası Check Constraint	128
Foreign Key Constraint	129
Var Olan Tabloya Foreign Key Constraint Eklemek	130
Rule ve Default	131
Constraint'leri İncelemek	131
Constraint'leri Devre Dışı Bırakmak	132
Bozuk Veriyi İhmal Etmek	132
Constraint'i Geçici Olarak Devre Dışı Bırakmak	133
<b>5 İLERİ SEVİYE SORGULAMA</b>	<b>135</b>
Alt Sorgu Nedir?	135
İç İçe Alt Sorgular Oluşturmak	136

Tekil Değerler Döndüren İç İçe Sorgular	137
Çoklu Sonuç Döndüren İç İçe Sorgular	139
Türetilmiş Tablolar	142
İlişkili Alt Sorgular	144
İlişkili Alt Sorgular Nasıl Çalışır?	144
SELECT Listesindeki İlişkili Alt Sorgular	144
WHERE Koşulundaki İlişkili Alt Sorgular	145
EXISTS ve NOT EXISTS	146
Veri Tiplerini Dönüşürtmek: CAST ve CONVERT	147
Common Table Expressions (CTE)	148
Rütbeleme Fonksiyonları ile Kayıtları Sıralamak	151
ROW_NUMBER()	151
RANK ve DENSE_RANK Fonksiyonları	152
RANK	152
DENSE_RANK	152
NTILE	153
TABLESAMPLE	154
PIVOT ve UNPIVOT Operatörleri	156
PIVOT	156
UNPIVOT	157
INTERSECT	159
EXCEPT	160
TRUNCATE TABLE ile Veri Silmek	160
İleri Veri Yönetim Teknikleri	161
Veri Ekleme	161
Sorgu Sonucunu Yeni Tabloda Saklamak	161
Stored Procedure Sonucunu Tabloya Ekleme	162
Sorgu Sonucunu Var Olan Tabloya Ekleme	164
Veri Güncelleme	164
Tabloları Birleştirerek Veri Güncellemek	164

Alt Sorgular İle Veri Güncellemek	165
Büyük Boyutlu Verileri Güncellemek	165
Veri Silme	167
Tablo Birleştirerek Veri Silmek	168
Alt Sorgular İle Veri Silmek	170
TOP Fonksiyonu İle Veri Silmek	170
Silinen Bir Kaydın DELETED İçerisinde Görüntülenmesi	171
Dosyaların Veritabanına Eklenmesi ve Güncellenmesi	171
OPENROWSET Komutu	171
FILESTREAM	173
FILESTREAM Özelliğini Aktifleştirmek	175
Mevcut Bir Veritabanında FILESTREAM Kullanmak	177
FILESTREAM Özelliği Aktif Edilmiş Bir Veritabanı Oluşturmak	179
Sütunları Oluşturmak	180
Veri EklemeK	181
Veri Seçmek	181
Veri Güncellemek	182
GUID Değere Sahip Sütun Oluşturulurken Dikkat Edilmesi Gerekenler	183
Veri Silmek	183
Verileri Gruplamak ve Özetlemek	184
Group By	184
Group By All	187
Having İle Gruplamalar Üstünde Şart Koşmak	188
Gruplamalı Fonksiyonlar (Aggregate Functions)	189
AVG Fonksiyonu	189
SUM Fonksiyonu	192
COUNT Fonksiyonu	194
MAX Fonksiyonu	195
MIN Fonksiyonu	197
Gruplanmış Verileri Özetlemek	198

CUBE	198
ROLLUP	200
GROUPING ile Özetleri Düzenlemek	201
<b>6 GEÇİCİ VERİLER İLE ÇALIŞMAK</b>	<b>203</b>
SQL Server Tarih / Zaman Veri Tipleri	203
DATE	203
Time	204
SmallDateTime	205
DateTime	206
DateTime2	207
DateTimeOffset	207
Girdi Tarih Formatları	208
SQL Server Tarih/Saat Fonksiyonları	210
GETDATE	210
CAST ve CONVERT ile Tarih Formatlama	211
FORMAT	213
FORMAT Fonksiyonunun Farklı Ülke	215
Para Birimleri İle Kullanımı	215
DATEPART	217
YY, YYYY ya da YEAR	218
QQ, Q ya da Quarter	218
MM, M ya da MONTH	218
DY, Y ya da DAYOFYEAR	218
DD, D ya da DAY	219
WK ya da WW	219
DW ya da WEEKDAY	219
HH ya da HOUR	219
MI, N ya da MINUTE	219
SS, S ya da SECOND	220
MS, MCS MILLISECOND	220

ISDATE	220
DATEADD	222
DATEDIFF	223
DATENAME	224
DAY	225
MONTH	225
YEAR	225
DATEFROMPARTS	226
DATETIMEFROMPARTS	226
SMALLDATETIMEFROMPARTS	226
TIMEFROMPARTS	227
EOMONTH	227
SYSDATETIME	228
SYSUTCDATETIME	229
SYSDATETIMEOFFSET	229
SWITCHOFFSET	230
TODATETIMEOFFSET	230
GETUTCDATE	231
<b>7 VIEW'LERLE ÇALIŞMAK</b>	<b>233</b>
View'ler Neden Kullanılır?	233
View Türleri	234
Alternatifler	234
View Oluşturmak	235
Kısıtlamalar	237
Gelişmiş Sorgular ile View Kullanımı	237
Tanımlanan View'leri Görmek ve Sistem View'leri	239
View'lerin Yapısını Görüntülemek	241
Sys.Sql_Modules	241
Object_Definition	242
Sys.SysComments	242

sp_helptext	243
T-SQL ile View Üzerinde Değişiklik Yapmak	243
View Tanımlamalarını Yenilemek	244
Kod Güvenliği: View'leri Şifrelemek	244
Schema Binding	246
View İle Verileri Düzenlemek: Insert, Update, Delete	246
View İle Verileri Düzenlemede Instead Of Trigger İlişkisi	247
JOIN İşlemi Olan View'lerde Veri Düzenlemek	248
WITH CHECK OPTION Kullanımı	249
İndekslenmiş View'ler	249
Parçalı View Kullanımı	252
Parçalı View Kullanılırken Dikkat Edilmesi Gerekenler	255
View'ları Kaldırmak	256
SSMS ile View Oluşturmak ve Yönetimi (Video)	256

## 8 İNDEKSLERLE ÇALIŞMAK 257

SQL Server Depolama	259
Veritabanı	259
Dosya	259
Extent	261
Page	261
Veri Page	262
İndeks Page	262
BLOB Page	262
GAM, SGAM, PFS	263
BCM	263
DCM	263
Page Split	264
Satırlar	264
İndeksler Nerelerde Kullanılır?	264
İndeksleri Anlamak	265

Clustered Indeks	265
Clustered Indeks Taraması (Scan)	266
Clustered Indeks Araması (Seek)	266
Non-Clustered Indeks	266
SQL Server Indeks Türleri	266
Unique Indeks	266
Sütuna Kayıtlı (Columnstore) Indeks	267
Parçalı Indeks	267
Eklenti Sütunlu Indeks	267
XML Indeks	267
Karma (Composite) Indeks	267
Kapsam (Covering) Indeks	268
Filtreli Indeks	268
Full-Text Indeks	268
İndeks Oluşturmak	268
İndeks Oluştururken Kullanılan İfadeler	269
Ayrıntılı Indeks Söz Dizimini Anlamak	269
ASC/DESC	269
INCLUDE	270
WITH	270
PAD_INDEX	270
FILLFACTOR	270
IGNORE_DUP_KEY	271
DROP_EXISTING	271
STATISTIC_NORECOMPUTE	271
SORT_IN_TEMPDB	272
ONLINE	272
ALLOW PAGE/PAGE LOCKS	273
MAXDOP	273
ON	273

İndeksler Hakkında Bilgi Edinmek	273
Unique Indeks Oluşturmak	275
Kapsam (Covering) Indeks Oluşturmak	276
Eklenti Sütunlu Indeks Oluşturmak	277
Filtreli Indeks Oluşturmak	277
İndeks Yönetimi	277
İndeksler Üstünde Değişiklik Yapmak	278
REBUILD: Indeksleri Yeniden Derlemek	278
REORGANIZE: Indeksleri Yeniden Düzenlemek	279
İndeksleri Kapatmak	279
İndeks Seçeneklerini Değiştirmek	280
İstatistikler	281
İstatistik Oluşturmak	281
İstatistikleri Silmek	282
<b>9 SCRIPT VE BATCH KULLANIMI</b>	<b>283</b>
Script Temelleri	283
USE İfadesi	284
Değişken Bildirimi	284
SET İfadesi Kullanılarak Değişkenlere Değer Atanması	285
SELECT İfadesi Kullanılarak Değişkenlere Değer Atanması	287
Batch'ler	287
Batch'leri Ne Zaman Kullanırız?	288
GO İfadesi	288
GO ifadesi ne işe yarar?	288
SQLCMD	289
AKİŞ KONTROL İFADELERİ	293
IF ... ELSE	294
ELSE Koşulu	295
İç İçe IF Kullanımı	297
CASE Deyimi	299

Söz Dizimi (Giriş Deyimi)	299
Söz Dizimi (Boolean Deyimi)	299
WHILE Döngüsü	301
BREAK Komutu	302
CONTINUE Komutu	302
WAITFOR İfadesi	303
WAITFOR DELAY	303
WAITFOR TIME	304
GOTO	304
<b>10 SQL CURSOR'LARI</b>	<b>307</b>
Cursor İçerisindeki SELECT Sorgusunun Farkları	308
Cursor'ler Neden Kullanılır?	308
Cursor'un Ömrü	309
Bildirim	310
Açılış	311
Kullanım / Yönlendirme	311
Kapanış	312
Hafızada Ayrılan Belleği Boşaltmak	312
Cursor Tipleri ve Özellikleri	312
Scope	312
Kaydırılabilirlik	313
FORWARD_ONLY Özelliği	313
SCROLLABLE Özelliği	313
Duyarlılık Kavramı	314
Cursor'lerle Satırları Dolaşmak: FETCH	314
FETCH NEXT	315
FETCH PRIOR	315
FETCH FIRST	315
FETCH LAST	315
FETCH RELATIVE	316

FETCH ABSOLUTE	316
TYPE_WARNING	316
Cursor Tipleri	317
Statik Cursor'ler	317
Anahtar Takımı ile Çalıştırılan Cursor'ler	322
Dinamik Cursor'ler	325
FOR <SELECT>	326
FOR UPDATE	326
<b>11 STORED PROCEDURE'LER</b>	<b>329</b>
Stored Procedure'lerin Faydaları	331
Stored Procedure Türleri	332
Extended Stored Procedure'ler	332
CLR Stored Procedure'ler	334
Sistem Stored Procedure'leri	334
Kullanıcı Tanımlı Stored Procedure'ler	334
Stored Procedure Oluşturmak	335
Stored Procedure İçin Gerekli İzin ve Roller	335
Stored Procedure İçin Kısıtlamalar	336
Stored Procedure'ü Çalıştırmak	336
NOCOUNT Oturum Parametresinin Kullanımı	336
Stored Procedure'lerde Değişiklik Yapmak	338
Stored Procedure'leri Yeniden Derlemek	339
Stored Procedure'ler İçin İzinleri Yönetmek	340
Stored Procedurelerde Parametre Kullanımı	341
Girdi Parametreler (Input Parameters)	341
Girdi Parametreler ile Stored Procedure Çağırırmak	342
Tablo Tipi Parametre Alan Stored Procedure'ler	342
Çıkış Parametrelerle Çalışmak (Output Parameters)	344
Çıkış Parametrelerini Almak	345
RETURN Deyimi	346

EXECUTE AS Modül Çalıştırma Bağlamları	347
EXECUTE AS CALLER	347
EXECUTE AS 'kullanıcı'	348
EXECUTE AS SELF	349
EXECUTE AS OWNER	349
WITH RESULT SETS ile Stored Procedure Çağırırmak	349
Stored Procedure Güvenliği	350
Stored Procedure'lerin Şifrelenmesi	350
Stored Procedure'ler Hakkında Bilgi Almak	352
Stored Procedure'lerin Kaldırılması	354
<b>12 T-SQL İLE HATA YÖNETİMİ</b>	<b>355</b>
Hata Mesajları	355
Mesajları Görüntülemek	359
Yeni Mesaj Ekleme	360
Parametreli Hata Mesajı Tanımlamak	362
Hata Oluştururken Kullanılabilecek Özellikler: WITH	363
WITH LOG	363
WITH NOWAIT	364
WITH SETERROR	364
Mesaj Silmek	364
Oluşan Son Hatanın Kodunu Yakalamak: @@ERROR	364
Stored Procedure İçerisinde @@ERROR Kullanımı	366
Hata Fırlatmak	368
RAISERROR İfadesi	368
THROW İfadesi	370
Hata Kontrolü ve TRY-CATCH	372
<b>13 DİNAMİK SQL NEDİR?</b>	<b>375</b>
Dinamik SQL Yazmak	375
EXEC[UTE]	375

EXEC içerisinde fonksiyonlar kullanılabilir mi?	380
EXEC ile Stored Procedure Kullanımı	380
Dinamik SQL Güvenlik Sorunsalı	381
EXEC Fonksiyonu İçerisinde Tür Dönüşümü	383
SP_ExecuteSQL ile Dinamik Sorgu Çalıştırmak	384
Dinamik SQL ile Sıralama İşlemi	386
SP_ExecuteSQL ile Stored Procedure Kullanımı	387
SP_ExecuteSQL ile INSERT İşlemi	388
SP_ExecuteSQL ile Veritabanı Oluşturmak	388
<b>14 KULLANICI TANIMLI FONKSİYONLAR</b>	<b>391</b>
Kullanıcı Tanımlı	392
Fonksiyon Çeşitleri	392
Skaler Kullanıcı Tanımlı Fonksiyonlar	392
Türetilmiş Sütun Olarak Skaler Fonksiyon	394
Tablo Döndüren Kullanıcı Tanımlı Fonksiyonlar	395
Satırda Tablo Döndüren Fonksiyonlar	395
Çoklu İfade İle Tablo Döndüren Fonksiyonlar	396
Kullanıcı Tanımlı Fonksiyonlarda Kod Gizliliği: Şifrelemek	400
Determinizm	401
Schema Binding	403
Tablolarla Tablo Tipi Fonksiyonları Birleştirmek	403
CROSS APPLY	405
OUTER APPLY	406
CROSS APPLY ve OUTER APPLY	406
Operatörlerinin Fonksiyonlar İle Kullanımı	406
Kullanıcı Tanımlı Fonksiyonların Yönetimi	408
Kullanıcı Tanımlı Fonksiyonları Değiştirmek	408
Kullanıcı Tanımlı Fonksiyonları Silmek	409

<b>15 SQL SERVER İLE XML</b>	<b>411</b>
XML	412
XML Veri Tipini Kullanmak	412
XML Tipi ile Değişken ve Parametre Kullanmak	413
Tip Tanımsız XML Veri İle Çalışmak (UnTyped)	414
Tip Tanımlı XML Veri İle Çalışmak (Typed)	417
XML Veri Tipi ile Çoklu Veri İşlemleri	419
XML Şema Koleksiyonları	421
DTD	421
DOCTYPE Bildirimi	421
ELEMENT Bildirimi	421
ATTLIST Bildirimi	422
XML Şema Koleksiyonları Hakkında Bilgi Almak	423
XML_SCHEMA_NAMESPACE ile Şema Koleksiyonlarını Listelemek	424
XML Şema Koleksiyonu Oluşturmak	425
XML Şema Koleksiyonu Değiştirmek	428
XML Şema Koleksiyonunu Kaldırmak	428
XML Veri Tipi Metodları	428
xml.query	430
xml.exist	432
xml.value	434
xml.nodes	435
xml.modify() ile XML Veriyi Düzenlemek	436
delete	437
replace value of	437
XML Biçimindeki İlişkisel Veriye Erişmek	438
FOR XML	438
RAW	439
AUTO	442
EXPLICIT	444

EXPLICIT ile Sütunları Gizlemek	448
PATH	449
OPEN XML	451
HTTP Endpoint'leri	454
HTTP Endpoint ve Güvenlik	455
HTTP Endpoint İle Kullanılacak	455
Veri Nesnelerinin Oluşturulması	455
HTTP Endpoint Oluşturulması ve Yönetilmesi	456
<b>16 SQL SERVER TRANSACTION</b>	<b>459</b>
Transaction ve Ortak Zamanlılık	460
Bölünemezlik (Atomicity)	461
Tutarlılık (Consistency)	461
İzolasyon (Isolation)	461
Dayanıklılık (Durability)	461
Transaction Bloğu	462
Transaction İfadelerini Anlamak	462
Transaction'ı Başlatmak: BEGIN TRAN	463
Transaction'ı Tamamlamak: COMMIT TRAN	463
Transaction'ı Geri Almak : ROLLBACK TRAN	463
Sabitleme Noktaları: SAVE TRAN	464
Transaction Oluşturmak	464
Sabitleme Noktası Oluşturmak: Save Tran	466
Try-Catch ile Transaction Hatası Yakalamak	468
Xact_State() Fonksiyonu	469
İç İçe Transaction'lar (Nested Transactions)	470
Ortak Zamanlılık ve İzolasyon Seviyeleri	471
Kilitleme (Locking)	471
Satır Versiyonlama (Row Versioning)	472
Ortak Zamanlı Erişim Anomalileri	472
Kayıp Güncelleme (Lost Update)	472

Tekrarlanamayan Okuma (Non-Repeatable Read)	473
Hayalet Okuma (Fantom Read)	473
Kirli Okuma (Dirty Read)	473
Kilitler	473
Kilitlenebilir Kaynaklar	474
Kilit Modları	474
Paylaşılmış Kilit (Shared Lock)	474
Özel Kilit (Exclusive Lock)	474
Güncelleştirme Kilidi (Update Lock)	474
Amaç Kilidi (Intent Lock)	475
Şema Kilitleri (Schema Locks)	475
Optimizer İpuçları ile Özel Bir Kilit Tipi Belirlemek	476
READCOMMITTED	476
READUNCOMMITTED/NOLOCK	476
READCOMMITTEDLOCK	477
SERIALIZABLE/HOLDLOCK	477
REPEATABLEREAD	477
READPAST	477
ROWLOCK	478
PAGLOCK	478
TABLOCK	478
TABLOCKX	478
UPDLOCK	478
XLOCK	479
Isolation Seviyesinin Ayarlanması	479
READ COMMITTED	479
READ UNCOMMITTED	480
REPEATABLE READ	480
SERIALIZABLE	480
SNAPSHOT	480

İzolasyon Seviyesi Yönetimi	481
Transaction Bazlı Snapshot İzolasyon	484
İfade Bazlı Snapshot İzolasyon	485
Kilitlenmeleri Yönetmek	485
Kilitlemeleri Gözlemelemek	485
Zaman Aşımını Ayarlamak	488
Kilitleme Çıkmazı: Deadlock	489
Aktivite Monitörü ile Kilitlenmeleri	489
Takip Etmek ve Process Öldürmek	489
<b>17 TRIGGER</b>	<b>491</b>
Trigger'ları Anlamak	491
Trigger'lar Nasıl Çalışır?	492
ON	492
WITH ENCRYPTION	492
WITH APPEND	493
NOT FOR REPLICATION	493
AS	493
FOR   AFTER İfadelerine ve INSTEAD OF Koşulu	493
Trigger Türleri ve INSERTED, DELETED Tabloları	493
INSERT Trigger	494
DELETE Trigger	494
UPDATE Trigger	494
Trigger Oluşturmak	494
Söz Dizimi (DML Trigger)	495
Söz Dizimi (DDL Trigger)	495
INSERT Trigger	495
DELETE Trigger	499
UPDATE Trigger	500
Birden Fazla İşlem İçin Trigger Oluşturmak	503
INSTEAD OF Trigger	504

INSTEAD OF INSERT Trigger	507
INSTEAD OF UPDATE Trigger	508
INSTEAD OF DELETE Trigger	509
IF UPDATE() ve COLUMNS_UPDATED()	514
UPDATE() Fonksiyonu	514
COLUMNS_UPDATED() Fonksiyonu	516
İç İçe Trigger (Nested Trigger)	516
Recursive Trigger	517
DDL Trigger'lar	518
Veritabanı Seviyeli DDL Trigger'lar	518
Sunucu Seviyeli DDL Trigger'lar	521
Trigger Yönetimi	522
Trigger'ı Değiştirmek	522
Trigger'ları Kapatmak ve Açmak	523
Trigger'ları Silmek	523
Veritabanı Seviyeli DDL Trigger'ları Silmek	524
Sunucu Seviyeli DDL Trigger'ları Silmek	524
<b>18 SORGU VE ERIŞİM GÜVENLİĞİ</b>	<b>525</b>
SQL Injection	525
SQL Injection Kullanımı	526
Stored Procedure ile SQL Injection	532
Saldırılara Karşı Korunma Yöntemleri	535
Karakter Filtreleyin	535
Kayıt Uzunluklarını Sınırlayın	536
Veri Tiplerini Kontrol Edin	537
Yetkileri Sınırlandırın	537
Uygulamaları Tarayın	537
Erişim Güvenliği	537
Erişim Güvenliğine Genel Bakış	538
İzinlerini Anlamak	538

Verilen İzinleri Sınamak	539
SQL Server Kimlik Doğrulama Yöntemleri	540
Windows Kimlik Doğrulaması	540
Karışık Güvenlik ve SQL Server Oturumları	541
Özel Amaçlı Oturumlar ve Kullanıcılar	541
Administrators Grubu ile Çalışmak	542
Administrator Kullanıcı Hesabı ile Çalışmak	542
sa Oturumu ile Çalışmak	542
NETWORK SERVICE ve SYSTEM Oturumları ile Çalışmak	542
Guest Kullanıcısı	542
dbo Kullanıcısı	543
sys ve INFORMATION_SCHEMA Kullanıcıları	543
Sunucu Oturumlarını Yönetmek	545
Oturumları Görüntülemek ve Düzenlemek	546
Kullanıcı, Oturum ID'si ve Parola	548
Parolanın Geçerlilik Süresi	549
Parola Uzunluğu ve Biçimi	550
Oturumlar Oluşturmak	550
SQL Oturumları Oluşturmak	552
Oturumları T-SQL ile Düzenlemek	553
Sunucu Erişimi Vermek ya da Kaldırmak	554
Oturumları Etkinleştirmek, Devre Dışı Bırakmak ve Kilidini Kaldırmak	555
Şifreleri Değiştirmek	556
Oturumları Kaldırmak	556
İzinler	557
İfade İzinleri	557
Nesne İzinleri	560
Roller	561
Sunucu Rolleri	561
Veritabanı Rolleri	563

Kullanıcı Tanımlı Standart Roller	563
Kullanıcı Tanımlı Uygulama Rolleri	563
Önceden Tanımlı Veritabanı Rolleri	563
Sunucu Rollerini Yapılandırmak	565
Oturum ile Roller Atamak	565
Birden Çok Oturuma Roller Atamak	566
<b>19 PERFORMANS VE SORGU OPTİMİZASYONU</b>	<b>567</b>
Performans Ayarı Ne Zaman Yapılmalıdır?	568
Donanım	569
I/O ve CPU Yoğunluğu ve RAM Kullanımı	570
Çökme Senaryoları	571
Online Sistemler Üzerinde SQL Server	573
Online Güvenlik	573
Online Performans	574
Online Destek	574
Sorgu Optimizasyonu	574
Minimum Kuralı	575
Geçici Tablolar	575
Filtreleme, Gruplama ve Sayfalama	576
İndeks	577
Filtreli İndeks (Filtered Index)	578
Sorgu Performansını Artırır	578
Bakım Maliyetlerini Düşürür	578
Disk Depolama Maliyetini Düşürür	578
View	579
Constraint	579
Trigger	580
Trigger'lar Reaktif'tir	580
Trigger'larda Ortak Zamanlılık	580
Trigger Genişliği	581

Trigger'lar ve Rollback	581
Performans İçin İstatistiksel Veri Kullanımı	581
Performans İçin DMV ve DMF Kullanımı	582
DMV ve DMF Hangi Amaç İle Kullanılır?	583
Bağlantı Hakkında Bilgi Almak	583
Session Hakkında Bilgi Almak	584
Veritabanı Sunucusu Hakkında Bilgi Almak	584
Fiziksel Bellek Hakkında Bilgi Almak	585
Aktif Olarak Çalışan İstekleri Sorgulamak	586
Cache'lenen Sorgu Planları Hakkında Bilgi Almak	588
Cache'lenen Sorgu Planlarının Nesne Tipine Göre Dağılımı	590
Parametre Olarak Verilen sql_handle'in SQL Sorgusunu Elde Etmek	590
Parametre Olarak Verilen plan_handle'in Sorgu Planını Elde Etmek	591
Sorgu İstatistikleri	592
İşlemlerdeki Bekleme Sorunu Hakkında Bilgi Almak	595
Disk Cevap Süresi Hakkında Bilgi Almak	596
Bekleyen I/O İstekleri Hakkında Bilgi Almak	598
DBCC SQLPERF ile DMV İstatistiklerini Temizlemek	599
Stored Procedure İstatistikleri Hakkında Bilgi Almak	600
Kullanılmayan Stored Procedure'lerin Tespit Edilmesi	602
Trigger İstatistikleri	603
Kullanılmayan Trigger'ları Tespit Etmek	605
Açık Olan Cursor'leri Sorgulama	606
<b>20 YEDEKLEMEMK VE YEDEKTEN DÖNMEK</b>	<b>609</b>
Veritabanını Yedeklemek	609
Yazılımsal Sorunlar	609
Donanımsal Sorunlar	610
Fiziksel Sorunlar	610
SQL Server Otomatik	610
Kurtarma İşlemi	610

Veritabanı Loglama Seçenekleri	610
Simple Recovery Model	611
Full Recovery Model	611
Bulk-Logged Recovery Model	611
Yedekleme Türleri	612
Tam Veritabanı Yedeği (Full Database Backup)	612
Fark Yedeği (Differential Backup)	612
Log Yedeği (Log Backup)	612
Yedekleme ve Kurtarma Planı Oluşturmak	613
Başlangıç Yedekleme ve Kurtarma Planlaması	613
Veri Ne Kadar Önemli?	613
Yedeklenen Veritabanlarının Türü Nedir?	613
Veri Ne Kadar Çabuk Kurtarılmalı?	614
Yedeklemeyi Gerçekleştirecek Donanım Var Mı?	614
Yedekleme İçin En Uygun Zaman Nedir?	614
Yedekleme, Sıkıştırılabilir Mi?	615
Yedeklemeleri Alan Dışında Saklamak Gerekıyor Mu?	615
Yedekleme Sıkıştırması Planlamak	615
Yedeklemeyi Gerçekleştirmek	616
SQL Server Management Studio ile Yedek Oluşturmak	617
T-SQL ile Veritabanı Yedeği Oluşturmak	618
T-SQL ile Transaction Log Dosyası	622
Yedeği Oluşturmak	622
SQL Agent ile Otomatik Yedekleme Planı Oluşturmak	623
Veritabanını Geri Yüklemek	628
T-SQL Geri Yükleme Komutlarını Kullanmak	629
<b>21 SQL SERVER MANAGEMENT OBJETS'İ KULLANMAK</b>	<b>631</b>
SQL Server Management Objects Uygulamaları	632
SMO ile Sunucu Bağlantısı Oluşturmak	634
Sunucu Özelliklerini Elde Etmek	635

Veritabanları, Dosya Grupları ve Dosyaların Listesini Almak	636
Veritabanı Özelliklerini Almak	637
Sunucudaki Tüm Veritabanlarını Listelemek	638
Veritabanı Oluşturmak	639
Veritabanı Yedeklemek	640
Veritabanı Geri Yüklemek	642
Veritabanını Silmek	643
Tablo ve Sütunları Listelemek	644
View Oluşturmak	645
Stored Procedure Oluşturmak	646
Bir Stored Procedure'ü Oluşturmak,	647
Değiştirmek ve Silmek	647
Veritabanındaki Tüm Stored	649
Procedure'leri Şifrelemek	649
Şema Oluşturmak	651
Şemaları Listelemek	652
Linked Server Oluşturmak	653
Oturum Oluşturmak	653
Oturumları Listelemek	654
Kullanıcı Oluşturmak	655
Kullanıcıları Listelemek	656
Rol Oluşturmak	657
Rolleri Listelemek	658
Rol Atamak	659
Assembly'leri Listelemek	660
Bir Tablonun Script'ini Oluşturmak	660

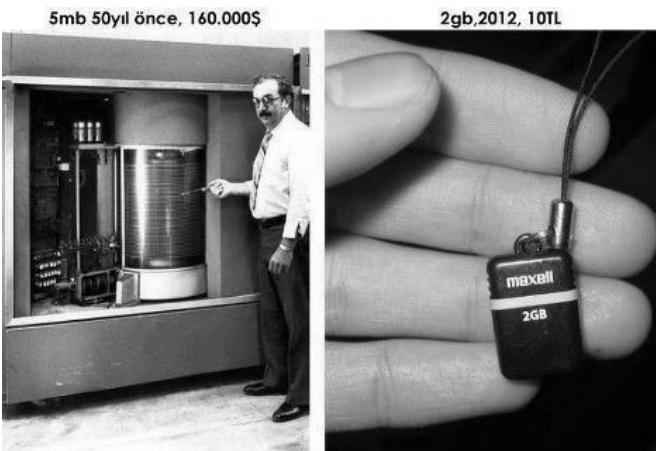
# VERİTABANI KAVRAMINA GİRİŞ

1

## VERİTABANI NEDİR?

Bilgisayar teknolojilerinin icadı ve gelişimiyle birlikte dijital veri deyimi sözcüklerimize eklendi. Dijital veri kavramı beraberinde depolama terimini de zorunlu hale getirdi. Bir şey var ise sanal ya da gerçek bir yer-alan kaplıyor demektir. Nasıl ki bir Hard Disk (*HDD*) masanın üzerinde dururken o masada yer kaplıyor ise, içerisindeki verinin de bir şekilde yer kaplaması kaçınılmazdır. Buna da sanal, yani dijital veri depolama alanı denir.

Bilgisayar sistemleri gelişikçe dijital veri kavramı da geliştirdi.



Bilgisayarın ilk yıllarda oda büyülüğünde olan bilgisayarlar, artık cebimize sığar hale geldi. Tabi ki depolama alanları da aynı şekilde. İlk zamanlarda 5 MB'lık bir depolama alanı devasa boyutlarda iken, artık kibrit kutusunun içine bir kaç tane sığabilecek kadar küçüldü. Bu veri depolama alanları büyükçe verinin yönetilmesi diye bir kavramın da oluşması kaçınılmaz hale geldi.

Bir dosya, doküman, yazı, kayıt bir şekilde depolanyor. Peki bu kayıtlara tekrar ulaşmak istediğimizde ne yapacağız?

Ticari ya da ticari olmayan sistemler, yazılımlar ve ihtiyaçları gelişikçe bunların yönetimi de zorlaştı ve bir programsal alt yapıya ihtiyaç duyulur hale geldi. Bu ihtiyacıa binaen geliştirilen ilk veritabanlarına örnek olarak, metinsel veri tutulabilen Notepad gibi doküman yönetimi yapılan programları örnek verebiliriz. Herhangi bir programlama dili ile dosyaya yazma-okuma yöntemleriyle kısıtlı ölçüde sorgulamalar yapıp, ekleme ve çıkarma işlemleri, sınırlı ölçüde düzenleme işlemleri yapılabilen metinsel veri yönetim araçları geliştirildi.

Ancak bilgisayar ve yazılım teknolojileri gelişikçe daha fazla veriyi, daha hızlı, güvenli ve mantıksal olarak depolama ihtiyacı doğdu ve böylelikle ilk veritabanı mimarisinin temelleri atılmış oldu.

Veriler veritabanında mantıksal bir düzen içerisinde saklanıp, çeşitli sorgular ve veri kayıt modelleriyle yönetilir hale geldi. Bu yazılımlar beraberinde yeni fikirleri ve ihtiyaçları getirdikçe veritabanı mimarisi de gelişti.

Bu veritabanında kayıtların sorgulama ve yönetimi işlemleri yapabilmek için veritabanı programlama geliştirildi.

## **SQL, T-SQL VE VERİTABANI PROGRAMLAMA**

Veritabanı programlama tekniği, ilk olarak matematiksel bir sözdizimine sahip olan **SQUARE** adlı bir sorgu dili ile geliştirildi. Bu teknolojinin daha kullanılması ve yaygınlaşabilmesi için matematiksel söz dizimli **SQUARE** dilinden vazgeçilerek, İngilizce'ye benzer sözdizimine sahip bir dil oluşturulmuş ve **SEQUEL** (*Structured English Query Language*) olarak adlandırılmıştır. Daha sonra da bu **SEQUEL** dili, global bir sorgu dili olması nedeniyle English kelimesi çıkarılıp, İngilizce söyleşisine paralel olarak **SQL** ile adlandırılmıştır.

İşte bir standart haline gelen **SQL** (*Structured Query Language*) Türkçesi, **Yapılardırılmış Soru Dili** olan veritabanı sorgu dilinin geliştirilmesinin sebebi de bu ihtiyaçları ortak bir mimari ile yapabilmekti. Veritabanını sorgulama işlemlerini yapan genel çalışmalar da **veritabanı programlama** denilmektedir.

SQL sorgulama dili bir **ANSI** (*American National Standards Institute*) standarıdır. Ve tüm veritabanı yönetim sistemleri bu standarı destekler. Ancak SQL sorgu dili yapı ve yetenek olarak basittir. Çok gelişmiş işlemler ve ileri yetenekler yapabilecek kapasitede değildir. Bu nedenle veritabanı motoru üreticileri, SQL dili standartlarını desteklemek şartıyla, kendi SQL sorgu motorlarını hazırlamaktadır.

Bu sorgu motorlarıyla ileri seviye işlemleri hızlı ve kolay şekilde yapacak sorgu cümlecikleri oluştururlar. Ve her sorgu motoru, kendine özgü geliştiricisi firmanın alt yapısında kullanılabilir. Örneğin; Oracle sorgu dili olan PL/SQL'i, SQL Server sorgu motorunda çalıştırılamaz. PL/SQL sorgu cümleleri SQL Server veritabanı sorgu motoru için sıradan cümleler anlamına gelir. Bu nedenle hata fırlatacaktır. İşte Microsoft da SQL Server için Transact-SQL açılımına sahip T-SQL adında yeni bir sorgu dili geliştirmiştir.

Bu yeni sorgu diliyle prosedür, trigger, hata yönetimi, sistem fonksiyonları, cursor gibi yüzlerce çeşit yeni yetenek kazandırmıştır. Bir veritabanı motorunun gücü sahip olduğu sorgu hızı, güvenlik ve diğer mimari yetenekleri haricinde sorgu dili yetenekleriyle de ölçülür diyebiliriz.

## İLİŞKİSEL VERİTABANI YÖNETİM SİSTEMİ

Verilerin tablolarda satır ve sütunlar halinde tutulduğu, yüksek veri tutarlılığına sahip, mimari yapısı ve index gibi özellikleri nedeniyle, sorgulama hızı yüksek veritabanı yönetimi için tasarlanan sistemdir. İlişkisel veritabanı bir veritabanı modelidir. Amacı veriye erişim ve hızı artırmadan yanında, veri tutarlığını sağlayıp gereksiz veri tekrarını da engellemektir.

**RDBMS** (*Relational Database Management Systems*), yani İlişkisel Veritabanı Yönetim Sistemleri olarak adlandırılan kavram, genel bir yazılım alanını belirtir. Bu yazılımlar arasında ücretli, ücretsiz, açık kaynak ya da kapalı kaynak olarak birçok farklı veritabanı yönetim sistemleri mevcuttur.

## SQL SERVER NEDİR?

Dünya'da bireysel ve kurumsal anlamda en çok tercih edilen, düşük maliyetli, yüksek performanslı, Windows işletim sistemi üzerinde çalışan, SQL standardını destekleyen, kendine özgü T-SQL sorgu diline sahip olan ve paket hizmet çözümleriyle, tam bir veritabanı yönetim sistemi çözümü sunan gelişmiş bir ilişkisel veritabanı yönetim sistemidir.

Gerçek anlamda yükseliş SQL Server 2005 ile birlikte gerçekleşen, SQL Server 2008 ve 2012 ile çok hızlı gelişim gösteren ve kısa sürede büyük firmalar tarafından tercih edilen bir **İVYS** olmuştur.

Bu gelişim ve yükselişinde Windows Server ailesi, Office çözümleri, Microsoft'ın geliştiricisi olduğu C#, VB.NET ve .NET teknolojisiyle birlikte tabii ki Microsoft'un **Cloud Computing** hizmeti olan **SQL Azure** ile **Windows Azure** gibi çözümlerin büyük payı vardır. Çünkü kurumlar tüm sorunlarını çözecek hizmetleri tek merkezden almayı tercih eder.

SQL Server hizmet paketinde bulunan **Reporting Services**, **Business Intelligence**, **Analysis Services** gibi birçok servis ve yönetim aracıyla birlikte gerçek bir veritabanı çözümüdür.

## NEDEN SQL SERVER?

İlişkisel Veritabanı Yönetim Sistemleri (**İVYS**) arasında kurumsal ve geliştirici dünyasında kabul gören ve kullanılan veritabanı yazılımlarının sayısı sınırlıdır. Açık kaynak ve ücretsizlere en çok kullanılan kategorisinde Oracle firması tarafından satın alınan MySQL ve açık kaynak olarak gelişmiş, güçlü bir alt yapı sunan PostgreSQL'i örnek gösterebiliriz.

Açık kaynak ve ücretsiz yazılım camiasında veritabanı kullanıcıları yani geliştirici ya da veritabanı yöneticileri genel olarak çok büyük hizmetlerde kullanmamak ile birlikte kendi çözümlerini kendileri üretmek zorundadırlar.

Örneğin; raporlama, sorgulama arayızları, IDE'ler, veritabanı çözümleri geliştirme ve istatistik gibi birçok alanda çeşitli üçüncü parti araçlar geliştirirler. Bu araçlar da ücretli ya da ücretsiz olarak dağıtılmaktadır.

Bu örnektenden de anlaşılacağı gibi veritabanı sadece sorgu çalıştırılan ve kayıt eklenen bir veri depolama aracı değil, aksine tamamen bu alandaki tüm araç

ve hizmetleri kapsayan bir hizmet bütünüdür. Ve böyle de olmak zorunluluğu vardır.

Veritabanı yazılımı kullanarak kurumsal, gizliliği yüksek, güvenlik öncelikli ve veri kaybı yaşamayı göze almak istemeyen kurumlar bu nedenle ücretsiz ya da açık kaynak veritabanı yazılımları kullanıp, profesyonel destek olmadan ve verilerini bir güvence altında tutmadan çalışarak böyle bir riske girmek istemezler. Bunun yerine ücretli ama maliyeti çok yüksek olmayan ve profesyonel destek alabilecekleri bir çözüm isterler.

Bu şekilde ücretli ve Dünya'da bu hizmeti en iyi şekilde verecek **IVYS (İlişkisel Veritabanı Yönetim Sistemleri)** yazılımlarından bazılarını SQL Server, Oracle, DB/2 şeklinde sıralayabiliriz.

Bu yazılımların da kendi içerisinde bazı ayırmaları ve farklı özellikleri mevcuttur. Örneğin, farklı platformlarda çalışması istenen, büyük veri hacimlerini işleyip yönetebilecek bir veritabanı yazılımı isteniyor ise çözüm Oracle gibi farklı platformlarda çalışan güçlü bir **IVYS** olacaktır. Ancak Oracle kullanıcıları genel olarak Linux tabanlı sistemlerde geliştirme ve yönetim sağladıkları için bu sefer gücü bir IT departmanı ihtiyacı doğabilir. Bu da çoğu firma için büyük maliyet anlamına gelir ve Oracle lisans bedeli olarak da maliyeti yüksek bir veritabanı yazılımıdır. Bu maliyetinin yanında kolay yönetilebilir, hata olasılığı düşük, geliştirme ve yönetim araçları kullanıcı dostu (*user friendly*) arayüzlere sahip bir çözümü sunma konusunda bazı sıkıntılar yaşanacağı aşikardır. Bu durumda Oracle doğru bir seçim olmayıabilir.

Ancak riski en aza indirmek, büyük verileri kolay ve hızlı yönetebilecek, yönetimi ve geliştirmesi kolay, geliştirme ve yönetim araçları kullanıcı dostu ve hata olasılığı düşük, kurulum ve konfigürasyonu kolay, lisans bedeli düşük, bir çok programlama dili ve platformıyla sorunsuz çalışabilen, geliştirme ve yönetimde kalifiye eleman bulma konusunda sıkıntı yaşatmayacak, IT masrafi düşük, profesyonel destek alınabilecek, işletim sistemi mimarisi üzerinde tam uyumlu bir veritabanı yazılımı isteniyorsa, bunun için SQL Server doğru seçimdir.

SQL Server 2000 sürümünde şu anki gibi güçlü ve profesyonel hizmet sunamamaktaydı. Ancak 2005, sonrasında 2008 ve 2012 ile Oracle'da bulunan birçok özellik, güçlü alt yapı ve mimari yetenek ile birlikte birçok yönetim

aracını hizmet paketine ekleyerek büyük bir gelişim gösterdi. SQL Server'ın geliştiricisi olan Microsoft firmasının işletim sistemi geliştirmesi ve gene SQL Server gibi geliştiricisinin Microsoft olduğu Windows Server ailesiyle çalışiyor olması, mimari açıdan %100 tam uyumlu ve ortak çalışabilen servis ve özellikleri, mimari olarak bütünlük olması, veritabanı uygulamaları geliştirmek için Microsoft' un geliştirdiği ve .NET destekli tüm diller ile sorunsuz ve hızlı geliştirme yapabilmesi, .NET alt yapısı ve C# ile VB.NET gibi programlama dillerinin geliştiricisinin de Microsoft olması ve Java, C++ gibi diğer .NET dışındaki diller ile de sorunsuz çalışabilmesi gibi daha birçok özellik nedeniyle genel bir hizmet paketi isteyen kurumlar için doğru bir seçim olmaktadır.

Hem sunucu ve istemci işletim sistemini, hem veritabanı yazılımını, hem programlama dilini, hem tasarım hem yönetim araçlarını, hem de programlama yapılabilen ofis araçlarını (MS Office) tek bir firmadan alarak destek masrafını azaltıp, tek mimari üzerinden çalışmayı kim istemez ki? İşte bu nedenle, Windows tabanlı çalışan kurumların tercihi SQL Server olmayı südürecektr.

## **SQL SERVER 2012 SÜRÜMLERİ VE ÖZELLİKLERİ**

### **SQL SERVER 2012 SÜRÜMLERİ**

SQL Server, kurumların ihtiyaçlarını karşılayacak şekilde mantıksal olarak birden fazla sürümme ayrılmıştır. Bir geliştirici ile orta ölçekli bir firmanın, orta ölçekli firma ile de büyük bir bankanın farklı veritabanı ihtiyaçları olacaktır. Microsoft, bu ihtiyaçları göz önünde bulundurarak farklı sürümler oluşturmuştur.

Bu sürümler SQL Server'ın gelişimiyle birlikte farklılık göstermiştir. Yeni yayınlanan SQL Server sürümünde kullanılan bir sürüm, sonraki versiyonda kaldırılmıştır. Ancak ana işlemler için hazırlanan sürümler değişmemektedir.

Şuan SQL Server 2012'de Standard Edition, Business Intelligence Edition ve Enterprise Edition olmak üzere üç ana sürüm var. Bu sürümlerin yanında ücretsiz olarak geliştiricilere sunulan Developer Edition ve kısıtlı özelliklere sahip Express Edition da SQL Server 2012 sürümünde yerini aldı.

SQL Server 2008'de var olan Datacenter Edition ve Workgroup Edition sürümleri SQL Server 2012'den kaldırılmıştır.

Bu sürümler içerisinde en gelişmiş olanı SQL Server 2008'de adı Datacenter Edition olan, SQL Server 2012'de ise Enterprise olarak değişen sürümdür. Business Intelligence Edition sürümü SQL Server 2012 ile tanıtığımız, iş zekası (business intelligence) uygulamalarına ihtiyaç duyulan, gene orta ve büyük işletmeler için hazırlanan sürümdür. SQL Server'ın temel sürümü olan Standard Edition ise temel ihtiyaçlara yönelik özellikleri içerir.

### **SQL SERVER 2012 EXPRESS EDITION**

SQL Server'ın kullanımı kolay, işlemci, ram ve veritabanı büyüklüğü gibi birçok mimari kısıtlamaları olan, küçük uygulamalar için kullanılabilecek ücretsiz sürümüdür. Bu sürümü ticari uygulamalarınızda da kullanmanızda herhangi bir sakınca yoktur. Genel olarak, küçük veritabanı kurulumu ve yönetimi gerçekleştirilecek yazılım uygulamalarında istemcilere kurularak yazılımin veritabanı ihtiyacını karşılamak için kullanılır.

### **SQL SERVER 2012 DEVELOPER EDITION**

Geliştiriciler için gelişmiş test ve uygulama alt yapısı olarak hazırlanan bu sürüm, SQL Server 2012'nin tüm özelliklerine sahiptir. Ancak sadece test ortamlarında kullanılmak üzere lisanslanan bir Enterprise Edition kopyasıdır. Test ortamı dışarısına çıkarılacağında Enterprise Edition sürümüne yükseltmek için yeniden lisanslanmalıdır.

### **SQL SERVER 2012 WEB EDITION**

Web ortamında kullanılacak özelliklere sahip şekilde hazırlanan bu sürüm, SQL Server 2012'nin hosting firmaları ya da kurum ve kuruluşların web sunucularında SQL Server ihtiyacını karşılamak için hazırlanmıştır.

### **SQL SERVER 2012 BUSINESS INTELLIGENCE EDITION**

SQL Server 2012 ve Visual Studio yeteneklerinin kullanılarak hazırlanan bir kurumsal iş zekası geliştirme sürümdür. Tarayıcı tabanlı veri arama ve görüntüleme gibi daha birçok özelliğe sahip bu sürümün Web, Developer ve Express sürümleri de bulunmaktadır.

### **SQL SERVER 2012 STANDARD EDITION**

SQL Server'ın değişmeyen ve ana sürümü niteliğindeki sürümdür. Küçük ve orta ölçekli kurum ve kuruluşlar için hazırlanan SQL Server'ın tüm ana özelliklerine sahip sürümdür.

## **SQL SERVER 2012 ENTERPRISE EDITION**

Büyük kurum ve kuruluşlar için büyük verilerin yönetim, depolama, işleme ve analiz etme gibi bir çok görevi en iyi ve performans ile gerçekleştirebileceğiniz SQL Server sürümüdür. Bu sürüm, büyük verilerin hızlı ve performanslı yönetilebilmesi için mimari olarak da en yüksek donanım ihtiyaçlarını destekleyecek şekilde tasarlanmıştır.

## **SQL SERVER 2012 COMPACT EDITION**

SQL Server'ın gömülü sistemler, mobil cihazlar için geliştiği sürümüdür. Genel olarak düşük kapasiteli sistemlerde kullanılır.

## **SQL SERVER LİSANSLAMA**

SQL Server birden fazla olan sürümü ile farklı çözümler sunuyor. Bu çözümler gibi lisanslama yöntemi de farklı şekillerde olabilmektedir.

SQL Server için temel olarak 3 lisanslama yöntemi bulunmaktadır.

- **Sunucu artı aygıt istemci erişim lisansı (Server plus device client access license - CAL)**

Bu modelde SQL Server'ın çalıştığı makine ile SQL Server'a erişen her cihazın (PC, PDA, mobil telefon, terminal vs.) lisanslanması gerekmektedir.

- **Sunucu artı kullanıcı istemci erişim lisansı (Server plus user client access license - CAL)**

Bu modelde SQL Server'ın çalıştığı makine ile SQL Server'a erişen her kullanıcının lisanslanması gerekmektedir.

- **İşlemci lisansı (Processor license)**

Bu modelde SQL Server'ın üzerinde çalıştığı işletim sistemi ortamındaki her bir işlemci için bir lisans gerekmektedir.

SQL Server 2012 ile aşağıdaki lisanslama seçenekleri sunulacak:

- **Enterprise(EE) sürüm** için çekirdek tabanlı lisanslama (**processor license**).
- **İş Zekası(BI) sürüm** için sunucu + CAL lisanslama.
- **Standart(SE) sürüm** için çekirdek tabanlı lisanslama (**processor license**) ya da sunucu + CAL lisanslama seçenekleri.

SQL Server 2012'nin çıkması ile aşağıdaki üç sürüm devreden çıkacaktır:

- **Datacenter:** Özellikleri Enterprise Edition sürümünde yer alacaktır.
- **Workgroup:** Temel veritabanı ihtiyaçları için Standard Edition sürümü kullanılacaktır.
- **Standard for Small Business:** Temel veri tabanı ihtiyaçları için Standard Edition sürümü tek sürüm olacaktır.

Bu lisanslama modelinde tek bir SQL Server CAL ile tüm sürümlere erişim imkanı olacak.

Microsoft tarafından değiştirilen ve sadeleştirilen SQL Server 2012 sürüm özelliklerini inceleyerek, hangi SQL Server 2012 sürümünün size uygun olduğunu inceleyerek bulabilirsiniz.

SQL Server satın alma ve lisanslama işlemleriyle ilgili gerekli bilgilere aşağıdaki bağlantılardan ulaşabilirsiniz.

<http://msdn.microsoft.com/en-us/library/cc645993.aspx>

<http://www.microsoft.com/sqlserver/en/us/get-sql-server/how-to-buy.aspx>

## **SQL SERVER KURULUMU & KURULUMU KALDIRMA**

### **SQL SERVER 2012 KURULUM GEREKSİNİMLERİ**

SQL Server 2012'nin doğru çalışabilmesi için öncelikle doğru kurulum ve eksiksiz bir alt yapı üzerine kurulum gerçekleştirilmelidir.

SQL Server 2012 kurulumuna başlamadan önce işletim sistemi uyumluluğunu kontrol etmemiz gereklidir. SQL Server 2012'nin çalışması için aşağıdaki işletim sistemlerinden birine ihtiyacı vardır.

- Windows Vista SP2 (*Ultimate, Enterprise, Business*)
- Windows 7 SP1 (*Ultimate, Enterprise, Professional*)
- Windows Server 2008 SP1 & SP2 (*Datacenter, Enterprise, Standard, Foundation*)

Eminim işletim sistemi listesinde klasikleşen Windows XP işletim sistemini aramışsınızdır. Ancak uzun süre önce Windows XP'ye destek vermeyeceğini açıklamıştı. Bu durumda tabi ki SQL Server 2012 için Windows XP desteği vermesi beklenemezdi.

İşletim sistemi seçimimizden sonra şu önemli alt yapıların sistemde kurulu olması gereklidir. Bunlar;

- Microsoft .NET Framework 3.5 SP1
- Microsoft .NET Framework 4.0
- Windows PowerShell 2.0

Windows PowerShell 2.0 Microsoft'un sistem yöneticileri için sunduğu komut satırı tabanlı yönetim aracıdır. Bu teknoloji kısaca PowerShell olarak bilinir. PowerShell, Windows Server 2008 R2 ve Windows 7 işletim sistemlerinde varsayılan olarak kurulu gelirken Windows Server 2008 ve Windows Vista işletim sistemlerinde manuel olarak kurulumu yapılmalıdır.

Microsoft .NET Framework 3.5 SP1 SQL Server kurulumu sırasında kurulmaz. Bu nedenle SQL Server kurulumuna başlamadan önce makineye kurulup, daha sonra bilgisayar yeniden başlatılarak, Microsoft .NET Framework 4.0 makineye kurulmalı (dilerseniz SQL Server kurulumu sırasında otomatik kurulabilir) ve bilgisayarı yeniden başlatıldktan sonra, SQL Server 2012 kurulumu için alt yapınız hazır hale gelmiş olacaktır.

## **SQL SERVER 2012 KURULUM & KURULUMU KALDIRMA İŞLEMLERİ**

SQL Server 2012 kurulum süreci oldukça basit bir araçtır. Kurulum sırasında diğer bir çok veritabanı yönetim sisteminde olduğu gibi herhangi bir parametre dosyası, path ayarı gibi kurulum ve bilgisayardan kaldırma işlemi sırasında ek yapılandırma işlemi gerektirmez.

Tabi ki SQL Server kurulumun bu kadar kolay olmasında en büyük etken, tek bir işletim sistemi platformunda çalışıyor olmasıdır. SQL Server sadece Windows platformunda çalışır. Windows ve SQL Server'in geliştiricisi de Microsoft'tur. Bu da iki yazılımlının birbiriyile sorunsuz ve %100 uyumlu çalışması anlamına gelir. İşte bu nedenle kurulum ve kaldırma işlemleri gayet basittir.



Bu bölümde sayfalarca ekran görüntüsü ile kitabın gereksiz şekilde doldurulması taraftarı değilim. Bu nedenle SQL Server 2012 kurulum ve kurulumu kaldırma işlemlerini, size verilen DVD eki içerisinde video anlatım olarak bulacaksınız.

## SQL SERVER ARAÇLARI

### BOOKS ONLINE

SQL Server'da birçok faydalı araç vardır. Bu araçların görevleri de kendilerine özgüdür. SQL Server Books Online, bir araç gibi görünmeyebilir. Ancak SQL Server'ı doğru ve tüm hatlarıyla anlatma görevi de Books Online aracına aittir.

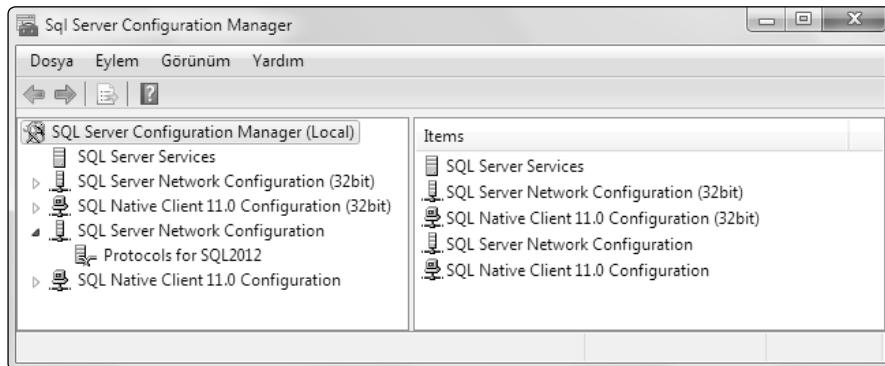
Bu nedenle hazırladığımız bu kitap ne kadar detaylı içeriğe sahip olsa da Books Online aracını mutlaka incelemenizi ve sürekli kullanmanızı öneririm.

SQL Server alt yapı çok geniş bir mimariye ve programlama gücüne sahiptir. SQL Server'ı ne kadar sık ve yoğun kullanırsanız, kullanım tamamını öğrenmeniz neredeyse mümkün değildir. Siz, rutin ve projelerinizde kullandığınız özellik ve güçlü noktalarını çok iyi kavrayın. SQL Server'ın diğer özelliklerine ihtiyaç duyduğunuzda ise araştırarak, bu kitabı ya da Books Online aracını kullanarak eksiklerinizi giderebilirsiniz.

İhtiyacınız olmayan bilgileri hafızanızda tutmak ya da tutmaya çalışmak sizi çok yorabilir ve asıl gerekli bilgileri öğrenebilmenizi daha zor hale getirebilir. Bu nedenle öğrendiğiniz bilgileri uygulayıp sonuçlarını gözlemledikten sonra unutmanız bir sorun teşkil etmeyecektir. Hatırlamanız gerekiğinde ufak bir araştırma ile tekrar aynı bilgileri kullanabilir hale geleceksinizdir. Bu kitabın ve Books Online'ın amacı öğretmek ve gerekiğinde unuttuğunuz bilgileri size hatırlatmaktır.

### SQL SERVER CONFIGURATION MANAGER

SQL Server Configuration Manager (*Yapilandırma Yöneticisi*), bilgisayar ve sistem yöneticileri için hazırlanmış hızlı, servis ve ağ yapılandırmasını gerçekleştirmeyi sağlayan bir yapılandırma aracıdır.



Bu isim ile SQL Server 2005 sürümünde duyuruldu ve bir çok farklı ayarın tek merkezden yönetilip yapılandırılmasını sağlar.

**Configuration Manager** iki ana yönetim sağlar. Bunlar;

- SQL Server Servis Yönetimi
- Ağ Yapılandırması

## SERVİS YÖNETİMİ

Servisler, **Configuration Manager** -> **SQL Server Services** içerisindeki sıralamaya göre şöyledir.

- **Integration Services:** Aynı ya da farklı platformlardaki verilerin taşınması ve dönüştürülmesi işlemlerini yapabildiğimiz Integration Service'lerini çalıştırır.
- **Analysis Services:** Analiz servislerini çalıştırır.
- **Full-Text Filter Daemon Launcher:** “Full-Text Search” için gerekli servisi çalıştırır.
- **Reporting Services:** SQL Server raporlama servisini çalıştırır.
- **SQL Server:** Veritabanının depolama, sorgulamalar ve diğer tüm veritabanı motorunun yapması gereken işlemleri gerçekleştirmek için kullanılan servistir. SQL Server'ın çalışması için bu servisin çalışıyor olması gerekmektedir.
- **SQL Server Agent:** SQL Server'da zamanlanmış görevleri yöneten servistir. Bu servisi kullanarak birden fazla ve farklı görevlere sahip zamanlanmış görevler oluşturabilirsiniz. Sizin belirleyeceğinizi gün ve saatte otomatik olarak bir veritabanının yedeğini almak bu görevlere bir örnek olabilir.

- **SQL Server Browser:** SQL Server'ın kurulu olduğu sistem tarafından tanınıp bulunabilmesi için gerekli servistir. Yerel ağ da server'ın bulunabilmesini sağlar.

## AĞ YAPILANDIRMASI

Sistem ya da bilgisayar yönetici olmayan, SQL Server'ın programsal geliştirmesini gerçekleştiren geliştiriciler için en önemli sistem sorunlarından biri SQL Server'a bağlanamama, ağ yapılandırmasında oluşan çeşitli sorunlar nedeniyle SQL Server ve bilgisayar ağı arasındaki uyuşmazlıklar gibi sistem yönetimi taraflı hatalardır.

Bu hataları daha iyi kavrayıp gerektiği zaman müdahale edebilmeniz için öncelikle SQL Server ağ yapılandırmasını ve kurallarını öğrenmeniz gerekmektedir.

SQL Server ağ yönetimini sağlayabilmek için NetLibs (*Net-Libraries*) adında ağ kütüphanelerine sahiptir. NetLibs'ler istemci uygulama ile ağ protokolü arasındaki iletişimini sağlar ve bağdaştırıcı gibi çalışır.

SQL Server 2012 ile sağlanan NetLibs'ler şunlardır;

- Shared Memory
- TCP/IP
- Named Pipes
- VIA

İstemci ve server'in birbiriyile ağ protokolü üzerinden iletişim sağlayabilmesi için her iki bilgisayarda da aynı NetLibs'lerin bulunması gereklidir. İstemci bilgisayarda server'da bulunmayan bir NetLibs seçilmesi, hataya sebep olacaktır. ('Specified SQL Server Not Found' hatası)

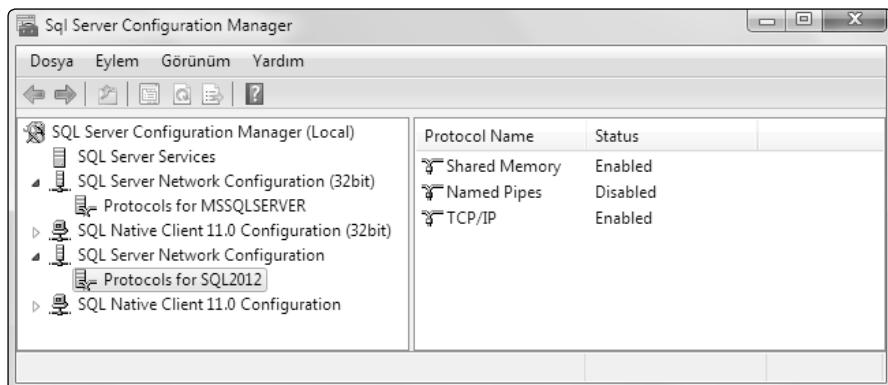
İstemci uygulamada kullanılan sürücü türü ODBC, OLE DB ya da SQL Native Client ne olursa olsun NetLib ile konuşacaktır.

Bir SQL Server sürücüsü ile NetLib'in haberleşme sırası aşağıdaki gibidir;

- İstemci uygulama sürücüsü (ODBC, OLE DB, SQL Native Client vs.) ile konuşur.
- Sürücü, istemci NetLib'i çağrıır.
- NetLib, uygun ağ protokolünü çağrıır ve veriyi server NetLib'ine iletir.
- Server NetLib'i, istekleri istemciden server'a iletir.

## PROTOKOLLER

SQL Server'ın hangi protokollerini dinleyebildiğini öğrenmek için SQL Server Configuration Manager aracımızdaki SQL Server Network Configuration sekmesini açarak, sağ bölümdeki protokoller listesinden tüm protokollerini görebilirsiniz.



SQL Server 2012 öncesi bir sürüm kurulu makinede protokollerden varsayılan olarak aktif olan Shared Memory'dir. Bu makine üzerine SQL Server 2012'de kurduğunuzda 2012 protokollerinde TCP/IP'nin de aktif olduğunu görebilirsiniz.

İstemcinin server ile iletişime geçip bağlantı kurabilmesi için server'ın, istemcinin kullandığı aynı port ile istemcinin kullandığı protokolü dinliyor olması gereklidir. Bu nedenle Named Pipes ortamındaysak, yeni bir kütüphane eklememiz gerekebilir. Bunun için, **Protocols** sekmesinde **Named Pipes** protokolü sağ tıklanarak **enable** seçeneğiyle aktif hale getirilmelidir.

*“Neden tüm NetLib'leri aktif etmiyoruz?”* diye sorduğunuzu duyar gibiyim. Tüm NetLib'leri aktif etmek bilgisayarda çalışan gerekliliğin olmaması gereksiz tüm servisleri başlatmak gibidir. Yani gerek olmadığı halde SQL Server'ı yavaşlatırsınız. Hem de güvenliğin üst düzey olması gereken SQL Server için gereksiz yere ağınıza açarak, bu şekilde güvenlik zayıflığına sebep olursunuz.

SQL Server'da desteklenen protokoller aşağıdaki gibi açıklanmıştır;

## TCP/IP

TCP/IP standart olan, en çok kullanılan ve mimari olarak ihtiyaç duyulan protokollerden biridir. SQL Server'a internet üzerinden erişmek için TCP/IP kullanılır. Çünkü IP kullanan tek protokol TCP/IP'dir.

SQL Server'ın interne açılması her ne sebepten olursa olsun güvenlik riskidir. Bu nedenle kritik veriler içeren bir SQL Server'i interne açarken bir kez daha düşünün. Farklı çözüm yolları var ise, o yolları denemeniz daha az sıkıntılı olacaktır. Eğer tek seçenekiniz SQL Server'i interne açmak ise, bu durumda güvenlik adımlarını iyi hesaplayarak, detaylı bir güvenlik çalışması yapmanız önerilir.

## NAMED PIPES

Named Pipes protokolü az kullanılan bir protokol olmasına rağmen TCP/IP'nin olmadığı, TCP/IP altındaki server'ları isimlendirmeye izin veren **DNS (Domain Name Service)** olmadığı durumlarda kullanılabilir.

SQL Server'a bilgisayar ya da sunucu adını vererek çalışabilirsiniz. Ancak DNS servisinin olmadığı durumda da sunucu adı, bilgisayar adı yerine **IP (Internet Protocol)** adresi vererek de çalışabilirsiniz.

## SHARED MEMORY

İstemci ile server'in aynı makinede kurulu olduğu durumlarda aradaki veri iletişimini hızlandırmak, performansı artırmak için kullanılır. İstemci, server'in veriyi sakladığı aynı eşlenmiş-bellek dosyasına doğrudan erişir. Bu yöntem ile veri gecikmesinin önüne geçerek iletişimi hızlandırır.

## VIA

Virtual Interface Adapter'in kısaltması olan **VIA**; iki sistem arasındaki bağlantıya özel, bir ağ arayüzüdür. Ve çok yüksek performansa sahiptir. Yüksek performanslı olmasının sebebi, özel olarak atanmış donanımdır.

## SQL SERVER MANAGEMENT STUDIO

SQL Server veritabanının programsal ve yönetimsel olarak kolay ve hızlı olmasında en büyük paha sahip aracı tüm bu işleri ve daha fazlasını gerçekleştiren SQL Server Management Studio aracıdır.



SQL Server’ı kullanabilmeniz için tabii ki bu aracı kullanmak zorunda değilsiniz. Farklı üçüncü parti araçlar mevcuttur. Ancak size önerebileceğim en iyi araç gene Management Studio’nun kendisi olacaktır.

Bu aracın detaylarıyla bu kitabın sayfalarını doldurmayı doğru bulmuyorum. Bu tür araçların özellik ve arayüzleri önemlidir. Ancak her versiyonda farklılık gösterebilen bu tür araçların ekran görüntüleriyle kitabın sayfalarını meşgul etmektense kitap için hazırladığım DVD ekinde bulunan videolardan daha etkin ve hızlı şekilde öğrenip bu konuyu kavrayabilirsiniz.

SQL Server Management Studio aracının özellikleri DVD içerisinde video eğitim serisi şeklinde anlatılmıştır.

## SQL SERVER PROFİLER

SQL Server veritabanı motoru biz sorgu çalıştırmasak da sürekli çok sayıda sorgu ile karşılaşır. Bunun sebebi; veritabanının yönetim ve istemcilerden, geliştirme araçlarından gelen isteklerin T-SQL diliyle veritabanına iletilmesidir. Biz araçları kullanırız ancak veritabanı motoruna giden yine T-SQL sorgularıdır.

Bu sorguları izlemenizi sağlayan aracın adı SQL Server Profiler’dir. Bu araç gerçek-zamanla sorgu izleme aracıdır. **Performance Monitor**, sistem yapılandırma konusunda meydana gelen olayların tamamını izlese de, SQL Server Profiler özel durumları izlemekle ilgilenebilir. SQL Server Profiler, nasıl yapılandırdığınıza bağlı olmakla birlikte, server’ınızda çalışan her sorgu cümlesinin özel T-SQL söz dizimini verir.

SQL Server Profiler, performans takibi, sorgu performansı, sorgu analizleri için veritabanı yöneticilerinin vazgeçilmez izleme aracıdır.

## SQL SERVER INTEGRATION SERVICES (SSIS)

Önceki sürümlerde **DTS** (*Data Transformation Services*) olarak bilinen **SSIS** bu araç SQL Server için karmaşık veri import ve export işlemleri için kullanılan

gerekli, güçlü ve gelişmiş bir entegrasyon aracıdır. **SSIS**, OLE DB ya da .NET veri sağlayıcıya sahip herhangi bir veri kaynağından veriyi alarak, SQL Server veritabanındaki tabloya gönderir.

## **SQL SERVER REPORTING SERVICES**

Reporting Services; SQL Server'ın rapor üretme, düzenleme ve çalışma platformudur. Web ortamında raporlar sunmak için yerleşik bir Windows Web Server ile birlikte çalışır. Raporlar, **Report Definition Language (RDL)** adı verilen XML tabanlı bir tanımlama dili kullanılarak hazırlanır. Business Intelligence Development Studio, hem basit hem de karmaşık raporlar için şablonlar kümlesi sağlar. Raporlar, Reporting Services tarafından talep edildiğinde, **RDL** dosyasına yazılır.

## **SQL SERVER BUSINESS INTELLIGENCE DEVELOPMENT STUDIO**

Business Intelligence Development Studio, SQL Server Business Intelligence'a özgü ek proje türleri içeren Microsoft Visual Studio uygulamasıdır. Business Intelligence Development Studio, Analysis Services, Integration Services ve Reporting Services projelerini içeren iş çözümleri geliştirmek için kullanacağınız birincil ortamdır. Her proje türü, karar destek çözümleri için gerekli nesnelerin oluşturulması için şablonlar sağlar ve nesnelerle çalışmaya yönelik çok çeşitli tasarımcılar, araçlar ve sihirbazlar sunar.

Business Intelligence Development Studio'nun menü ve özellikleri Visual Studio versiyonu ve kurulu olup olmamasıyla belirlenir. Visual Studio kurulu ise Visual Studio menüleri ve SQL Server Analysis Services ile ilişkili menüler ve şablonları karışımı olacaktır.

Özetlemek gerekirse; bu araç bize analiz (*Analysis Services*), raporlama (*Reporting Services*), Integration Services paketleri gibi birçok iş geliştirme araçlarını kullanabilmemiz için çeşitli araç, şablon ve menülerden oluşan bir geliştirme aracıdır.

## **BULK COPY PROGRAM (BCP)**

**Bulk Copy Program (BCP)** komut satırı aracıdır. Bu araç, SSIS geliştirilmeden önce SQL Server'ın import/export işlemlerini gerçekleştirirdi. **BCP**, halen

kullanılır ve bu aracı kullananların başında SQL Server'ın kendisi gelmektedir. Kurulum sırasındaki işlemlerin komut satırından hızlı olarak gerçekleşmesini sağlar.

## **SQLCMD**

Microsoft'un komut satırı kullanmayı seven geliştiriciler ve sistem yöneticileri için geliştirdiği komut satırı aracıdır. Bu aracın diğer bir adı da OSQL olarak bilinir.

Script'lerin çalıştırılması için kullanılan bu araç işini çok iyi yapabilir. Ancak genel olarak bu aracın yaptığı işlemleri yapan birçok SQL Server aracı vardır.

## **ÖZET**

Bu bölümde anlatılan araçlar SQL Server için faydalı ve her biri kendi alanında uzmanlık seviyesinde gerekli araçlardır. Ancak genel olarak en çok kullanacağınız araç SQL Server Management Studio'dur.

Tabi ki tüm araçları kullanmayabilirsiniz. Ancak gerçek bir uzman öncelikle takım çantası (geliştirme araçları) geniş ve seçenekleri çok olandır. Bu nedenle bu araçlardan haberdar olmanız, nasıl kullanıldığını bilmenizde ve burada anlatılmayan diğer araçları da zaman içerisinde araştırıp öğrenmenizde fayda vardır.

# **SQL SERVER VERİTABANI NESNELERİ**

SQL Server üzerinde her şey birer nesnedir demek hiç de yanlış olmaz. Bu nesneler veritabanı mimarisini oluşturur. Her nesneyi yap-boz'un bir parçası olarak düşünürsek bunlar bir araya getirildiğinde tam bir veritabanı oluşmaktadır.

Bu bölümde veritabanını oluşturan mimarının parçalarını inceleyerek işe başlıyoruz. Bu parçaları öğrendiğinizde veritabanı yapısını programatik olarak daha iyi anlayacak ve ileriki konularda iyi bir alt yapı sahip olacaksınız.

## Bu Bölümde Neler Öğreneceğiz?

Veritabanının yap-boz gibi nesnesel parçalardan oluştuğunu belirtmiştık. Bu mimarinin oluşmasının temeli veritabanının gerekliliğini zorunlu hale getiren veri kavramıdır.

SQL Server sadece bir veritabanı değildir. Yani sadece güvenli şekilde verilerin saklandığı, ekleme, güncelleme, silme gibi işlemlerin yapıldığı bir mimari değildir. SQL Server bir hizmet çözümü paketidir.

SQL Server'da amaç; sadece depolama işini yapmak değil geliştirilen birçok servis ile bu verinin içeri ve dışarı aktarılmasını sağlamak, güvenlik, raporlama, istatistik, analiz, performans, iş zekası ve yönetimi gibi birçok farklı görevi tek bir çözüm olarak sunmaktadır.

Bu bölümde SQL Server veritabanı mimarisini oluşturan nesneleri inceleyerek mimarinin bütününe görmeye çalışacağız.

## VERİTABANI NESNELERİNE GENEL BAKIŞ

SQL Server ilişkisel veritabanı yönetim sistemi birçok nesne içerir. Bu nesnelerden bazıları şunlardır;

Database	CLR Assembly	User-Defined Function
Index	Filegroup	User-Defined Data Type
Table	Diagram	View
Transaction Log	Full-Text Catalog	Trigger
Rule	User	Stored Procedure

## DATABASE NESNESİ

Veritabanı nesnesi SQL Server ile ilişkili en önemli, temel ve en yüksek seviyeli nesnedir. Bu bölüm sonunda anlayacağınız bir gerçek şu ki; diğer nesnelerin birçoğu veritabanı nesnesinin elemanlarıdır. Yani veritabanı nesnesi olmadan bahsedeceğimiz nesnelerin birçoğu anlamsız kalacaktır. Çünkü bu nesnelerin çoğu bir veritabanı nesnesi üzerinde çalışmak için tasarlanmıştır.

Veritabanı, nesneler içerisinde en temel nesnelerin başında gelen tablo ve view, stored procedure, function, trigger gibi birçok nesneyi içinde barındıran bir nesnedir.

SQL Server ilk yüklendiğinde dört sistem veritabanı ile birlikte kurulur.

- master
- model
- msdb
- tempdb

Bu veritabanları SQL Server'ın doğru çalışması için gereklidir. Eksikliklerinde SQL Server doğru çalışmayaçaktır.

## **MASTER**

**master** veritabanı, sistemin bütünüyü izlemeyi sağlayan, sistem tabloları kümesini içine alır. Sahip olduğu sistem prosedürleri ve view'ler ile veritabanı işlemlerinin, tablo ve diğer nesneler ile ilgili bilgi alma trafiğinin beyını oluşturuyor da diyebiliriz.

Örneğin;

**CREATE DATABASE DIJIBIL** kodunu çalıştırarak oluşturacağınız;

	name
1	master
2	tempdb
3	model
4	msdb
5	ReportServer\$SQL2012
6	ReportServer\$SQL2012TempDB
7	AdventureWorks2008R2
8	DIJIBIL

**DIJIBIL** isimli veritabanı **master** veritabanında **Views -> System Views** içerisinde **sys.databases** view'ini çalıştırarak tüm veritabanlarını listeleyecektir.

---

```
SELECT * FROM sys.databases
```

---

Server'ı tanımlayan her şey **master** veritabanında saklanır. Bu nedenle içerdiği veriler kritiktir ve silinemez.

**master** veritabanındaki sistem tabloları, view ve prosedürler genel olarak veritabanı yöneticileri tarafından kullanılsa ya da onların ihtiyaçlarını görecek

şekilde hazırlanmış olsa da geliştiriciler için de performans analizi ve bilgi almak için gereklidir.



Microsoft sistem tablolarını kullanmanın tehlikeli olduğunu bildirir ve sadece son üç versiyon için sistem tablolarının kullanımını önerir. Unutmayalım ki, SQL Server gelişikçe veritabanı yapısı, nesneler ve `master` veritabanı yapısında da değişiklikler olabilir. Hatta Microsoft, `master` veritabanının her versiyonda değişikliğe uğradığının garantisini veriyor da diyebiliriz. Bu nedenle bu tablolarda değişiklik yapmamanız kesinlikle önerilir. Küçük bir fayda sağlamak için yapacağınız değişiklik SQL Server'ın çalışmamasına sebep olabilir.

## MODEL

`model`, veritabanı oluşturma işleminde referans alınmak için hazırlanan bir `model` veritabanıdır. Yeni bir veritabanı oluştururken `model` veritabanı yeni bir şablon oluşturur. Yani isterseniz yeni oluşturacağınız veritabanlarının nasıl olacağını, `model` veritabanında değişiklik yaparak belirleyebilirsiniz. Örneğin; sistem üzerinde oluşturulan her yeni veritabanına, kopyalanan kullanıcı grupları oluşturabilirisiniz. `model` veritabanı, diğer veritabanları için şablon olduğu için gerekli bir veritabanıdır ve silinemez.

Oluşturduğunuz herhangi bir veritabanı `model` veritabanının büyüklüğünde olmalıdır. Yani `model` veritabanı 100 MB ise oluşturacağınız veritabanı bu büyülükten küçük olamaz. Bu nedenle varsayılan ayarları değiştirirken her zaman dikkatli olmanız önerilir.

## MSDB

`msdb` veritabanı, SQL Agent'ın herhangi bir sistem görevini sakladığı yerdir. Veritabanı üzerinde herhangi bir zamanlayıcı, örneğin yedekleme zamanlayıcısı gibi ya da stored procedure'un ilk çalışması ile ilgili kayıtlar da `msdb` de tutulur.

## TEMPDB

`tempdb`, SQL Server'da gerçekleştirilen sorgulama işlemlerine geçici hafıza görevi gören `temporary database`'dır. Önemli bir veritabanıdır. Karmaşık ya da geniş bir sorgulama yaptığınızda, SQL Server geçici tablo oluşturması gerektiğinde bu işlemleri `tempdb` ile yapar. Aynı zamanda T-SQL ile geçici tablo oluşturduğunuzda ya da geçici veri depolama işlemlerinde bu verilerin tutulduğu yer `tempdb`'dir.

**tempdb** veritabanının sadece içeriği veri ya da nesneler geçici değil, veritabanının kendisi de geçicidir. SQL Server her başlatıldığında **tempdb** sıfırdan yeniden oluşturulur.

## ADVENTUREWORKS

Tüm veritabanı yönetim sistemlerinde geliştiricilerin kullanacağı örnek veritabanı tasarımlarının olması örnek çalışmaları açısından faydalı olmaktadır. SQL Server eski sürümlerinde **Northwind** veritabanı örnek şeması vardı. Ancak SQL Server özelliklerinin büyük yenilikler içermesiyle bu tür eski örnek şemalar yeterli olmamaya başladı. Microsoft, SQL Server'ın gücünü temel özellikleriyle tam olarak yansıtacak bir veritabanı şeması hazırlamak istiyordu. **AdventureWorks** bu düşüncenin bir ürünüdür. Mimari olarak detaylı ve büyük bir alt yapı olarak hazırlanmıştır.

Aşağıdaki adresten bu veritabanının eski ve yeni sürümlerine ulaşabilirsiniz.

<http://msftdbprodsamples.codeplex.com>

## ADVENTUREWORKSDW

**AdventureWorksDW**, (*DW, Data Warehouse = Veri Ambarı*) Analysis Service için hazırlanan örnek veritabanı şemasıdır. Microsoft'un transaction veritabanı örneği ile analiz örneğinin birlikte çalışmasını gösteren bir örnektir.

## TRANSACTION LOGS (İŞLEM GÜNLÜKLERİ)

Veritabanı dosyası verilerin kaydolduğu ve okunduğu dosyalardır. Ancak bu işlemin daha doğru ve düzgün çalışabilmesi için farklı bir mimari geliştirilmiştir. Öncelikle veritabanında gerçekleşecek herhangi bir işlem direkt olarak veritabanı dosyasına yazılır. Önce değişiklikler transaction log dosyasına yazılır. Zaman olarak bir sonraki adımda, veritabanı bir checkpoint yaparak, log dosyasına yazılan değişikliklerin gerçek veritabanı dosyasında üretildiği zaman düzenler.

Veritabanı dosyası rastgele, fakat log dosyası sırasıyla yazılmıştır. Veritabanı dosyasının rastgele yapısı hızlı erişim imkanı kazandırırken, log dosyasının sıralı yapısı değişiklikleri sırasıyla izleyebilme imkanı sağlar. Veritabanı üzerinde çalıştırılan ve gerçekleştiği varsayılan işlemler log dosyasında biriktirilir ve sonra server bu değişiklikleri fiziksel olarak veritabanı dosyasına gerçekleşen zaman dilimiyle yazar.

## FILEGROUPS (DOSYA GRUPLARI)

SQL Server'da tüm tablolar ve veritabanı ile ilgili her şey tek bir veritabanı dosyasında saklanır. Bu dosya **Primary Filegroup** adı verilen birincil dosya grubu üyesidir. Bununla birlikte bu düzenlemeye bağlı olmak zorunda değilsiniz.

SQL Server pek sınır sorunu olmamakla birlikte 32.000' in üzerinde **Secondary Files** denilen ikincil dosyalar tanımlayabilirsiniz.

Sadece bir tek **Primary Filegroup**, yani birincil dosya grubu olmakla birlikte, 255 adet ikincil dosya grubu oluşturabilirsiniz.

İkincil dosya grubu `CREATE DATABASE` ya da `ALTER DATABASE` komut seçenekleriyle oluşur.

## DIAGRAMS (DİYAGRAMLAR)

Tablolar aslında SQL kodları ile oluşturulan yapılandırır. Ancak görsel olarak da daha iyi anlayıp yönetebilmemiz adına tablo tasarıımı denen bir olgu ile çalışırız. Diyagram'lar da bu tablo tasarımı olusunun bir adım daha ötesidir diyebiliriz. Bir tablonun içeriğini ve var ise başka tablolar ile ilişkilendirmelerini gösteren ve yönetebileceğimiz bir veritabanı görselleştirme aracıdır.

Veritabanı programlama alanında uğraşacak geliştiricilerin sık göreceği yapı Entity-Relationship kısa adıyla ERD'dir. ERD diyagramda veritabanı iki parçaya ayrılmıştır. Varlıklar (üretici, ürün gibi) ve ilişkiler(islenecek malzemeler ve satılacaklar) gibi.

SQL Server gelişikçe geliştirme ve modelleme araçları da gelişmektedir. Bu nedenle her yeni sürümde SQL Server Diagram Designer'a yeni özellikler eklenip kullanımı geliştirilmektedir. Bu tasarım aracı varlık-ilişkili modeli tam olarak yansıtamasa da veritabanı modellenmesinde etkili bir araçtır.

## SCHEMAS (ŞEMALAR)

Veritabanı mimari olan çokça nesneye ve katmana sahiptir. Bunların nesnesel olarak isimlendirilebilmesi için mantıksal bir isimlendirme yapısına ihtiyaç vardır. Schema'lar veritabanı ile içerisindeki nesnelerin mantıksal isimlendirmesini yapar.

SQL Server'da varsayılan Schema adı **dbo**, yani **Database Owner = Veritabanı Sahibi** anlamına gelir. Her kullanıcı varsayılan Schema'ya sahiptir. SQL Server tüm nesneleri otomatik olarak varsayılan Schema içerisinde arar.

Eğer ulaşılmak istenen nesne varsayılan Schema içerisinde değilse, **schema\_ad.nesne\_ad** şeklinde nesnenin içinde bulunduğu Schema adını da nesnenin başında belirterek yazmamız gereklidir.

Schema yapısı doğru ve yerinde kullanılmadığı takdirde karmaşa ve birçok sorun getirir. Bu nedenle özel durumlar haricinde kullanılması gereklidir.

## TABLES (TABLOLAR)

Bir veritabanının en temel yapısı tablolardır. Veritabanı nesnelerinin neredeyse tamamı bir tablo ya da içerisindeki veriyle alakalı işlem yapmak için tasarlanmıştır. Tablolar satır ve sütunlardan oluşur. Alan veri(sütunlar), tabloda saklanacak verilerin her biri için bir alan oluşturmak ve bu alanlara yapılacak verinin tipini belirtmeye yarar. Kayıt veri (satırlar), belirlenen alanlara verilerin girilmesiyle oluşur. Satırlar girilirken sütunların veri tipleri ve kısıtlamaları kontrol edilir. Bu kontrol işlemini gerçekleştirmek için tabloların yapısal bilgilerini tutan metadata'lar kullanılır.

## INDEXES (İNDEKSLER)

Index, sadece tablo ve view için bulunan bir nesnedir. Index sıralanma işlemini daha disiplinli yaparak performans artırmayı ve hızlı sorgu sonucu döndürmeyi amaçlar.

Index kavramı denince akla ilk gelen örnek kütüphane sistemleridir. Bir kütüphane bulunması gereken bir kitabı içerisindeki on binlerce kitabı tek tek inceleyerek bulmak yerine, bir index hazırlanarak belli bir arama ve sıralama algoritması üzerinden daha kısa sürede bulunması sağlanır.

En basit sıralama algoritması hepimizin bildiği, eski telefon rehberlerindeki A' dan Z'ye sıralama yaparak arama işlemini, bir nebze kolaylaştırıp hızlandıran index'leme yöntemidir.

Index'ler iki gruba ayrılır.

- **Clustered:** Tablo başına sadece bir clustered index kullanılabilir. Clustered index ile index'lenen tablo, fiziksel olarak bu index ile sıralanır.

Örneğin; bir ansiklopedi dizini hazırlıyorsanız, ansiklopedideki bilgiler sayfa numarasına göre sıralanacaktır.

- **Non-Clustered:** Tablo başına birden fazla non-clustered index tanımlanabilir. Non-clustered index mimarisinde fiziksel bir sıralama söz konusu değildir. Yani, clustered index gibi verinin kendisini değil pointer’ını tutmaya yarar.

## **CONSTRAINTS (KISITLAMALAR)**

Constraint’ler tablo sınırları içinde bir nesnedir ve adından da anlaşılacağı gibi tablo üzerinde bazı kısıtlamalar ile veri bütünlüğünü sağlamayı amaçlar.

## **VIEWS (GÖRÜTÜMLER)**

View bir tablonun sanal görüntüsüdür. Gerçekte içerisinde bir veri olmayan, ancak içeriğinde belirtilen SQL sorgu bloğunun getireceği verileri kendisine verilen kısa bir ad ile kullanıcıya tablo olarak göstermek için tasarlanmıştır. Bir ya da daha fazla tablodan veri çekerek tek sorgu sonucu olarak döndürebilir. Bu işlemi yapacak sorgu planı view içerisinde saklanır.

View’ler genel olarak birkaç nedenle kullanılır. Bunlar; güvenlik, performans ve kullanım kolaylığıdır. Hazırlanacak sorguların sonuçlarının tamamının görünmesini istemeyebiliriz. Örneğin; tablonun tamamının görünmesi veritabanı güvenliği açısından bir risk olabilir. Ancak hazırlanacak bir view’ in kullanımı sağlayabilir ve tablonun sadece belirlenen sütunlarını view içerisinde çağırarak bir kısıtlama yapabiliriz. Bu da veri erişim güvenliğini artıracaktır. Aynı zamanda performans ve kullanım kolaylığı gibi tüm faydaları ileride anlatılacak view bölümünde tüm detaylarıyla anlatılacaktır.

## **STORED PROCEDURES (SAKLI YORDAMLAR)**

Storec procedure, diğer adıyla sproc’lar SQL Server programlama (T-SQL) alt yapısının en önemli ve çok kullanılan özelliklerinden biridir. Sproc’lar veritabanı içerisinde yapılacak işlemlerin bir program düzende gerçekleştemesini sağlayan T-SQL programcılarıdır.

İçerisinde T-SQL gücünü kullanarak değişkenler, girdi ve çıktı parametreleri, hata yakalama, klasik sql sorguları gibi birçok özelliği kullanarak işlemler yapılabilir. Sproc’lar güvenlik, performans ve hızlı işlem gerçekleştirebilme gibi birçok faydaya sahiptir.

- Sproc'lar, server başlatıldığından derlenir (*compiling*) ve derlenmiş olduğu için yeniden kullanımında zamandan tasarruf sağlar.
- Sproc'lar kısa bir isim ile adlandırılırlar. Ve içerisindeki veriyi şifreleme özelliği de bulunduğu için dilerken T-SQL sorgularınızı sproc'un kullanıcılarından gizleyerek veritabanı güvenliği açısından ek fayda sağlayabilirsiniz.
- Bir sproc başka bir sproc tarafından çağrılabılır. Yani iç içe kullanılabilir. Bu da daha az kod ile daha çok işlem yapmayı ve fonksiyonel bir veritabanı programlamayı sağlar.
- .NET tabanlı diller ile SQL Server içerisinde sproc geliştirilebilir.
- Sproc'lar yazılım projesi içerisinde ADO.NET ya da benzeri veri katmanları tarafından direk olarak kullanılabilir. Bu da yazılım içerisinde herhangi bir T-SQL kodu yazmamanızı sağladığı gibi, sproc kod bloğu içerisinde herhangi bir değişiklik yapılacak ise, veritabanında bulunan sproc'un **ALTER PROCEDURE** ile güncelleme yapılarak gerçekleştirilebilmesini sağlar.

## **TRIGGERS (TETİKLEYİCİLER)**

Trigger'lar (*tetikleyici*) tablolar üzerinde gerçekleştirilen ekleme, güncelleme, silme ve kopyalama gibi işlemler gerçekleştikten sonra ya da önce gerçekleşmesi istenen işlemlerin otomatik olarak gerçekleşmesini sağlayan T-SQL programcıklarıdır.

SQL Server ve diğer veritabanı yönetim sistemlerinde önemli bir yere sahip olan trigger'lar, hızlı, yönetilebilir ve profesyonel veritabanları için sproc'lar kadar olmasa da sık kullanılan nesnelerdir.

## **USER-DEFINED FUNCTIONS**

### **(KULLANICI TANIMLI FONKSİYONLAR)**

UDF, Türkçe anlamı Kullanıcı-Tanımlı Fonksiyonlar sproc'lara çok benzer. Hatta arasında çok az fark var diyebiliriz.

UDF'lerin Sproc'lara göre bazı farklılıklar;

- UDF'ler yardımcı programcıklardır ve sproc'lar gibi güçlü ve veritabanı üzerinde sistematik bir değişiklik, e-mail gönderimi, tabloların değişimi, sistem ya da veritabanı parametrelerinin değiştirilmesi gibi bir işlem gerçekleştiremez.

- UDF'ler image, cursor, timestamp, text ve ntext tipleri haricinde SQL Server veri tiplerinin bir çoğuna değer döndürür.

UDF'leri programlama dillerindeki fonksiyonlar gibi düşünüebiliriz. Dışarıdan birden fazla parametre ile değer alır ve bu değerleri belirlenen işlemlere tabi tutulur ve sonucunda da geriye bir değer döndürür. Bu şekilde veritabanında fonksiyonel programlama yapmamıza yardımcı olur.

## **USER-DEFINED DATA TYPES (KULLANICI-TANIMLI VERİ TIPLERİ)**

Kullanıcı-Tanımlı veri tipleri SQL Server içerisinde özel veri tipi oluşturup kullanmanızı sağlar. Bu veri tipleri temel olarak sistem veri tipleri gibidir.

Bu özellik teknik olarak sunulsa da kullanırken dikkatli olmanız gerekmektedir. Bir nevi verinizi, veritabanınızı SQL Server'in sisteminde olmayan veri tipiyle geliştiriyor olmanız, geliştirici olarak sorumluluğunuza artırdığı gibi hata ve olası riskleri de artırmaktadır.

### **DEFAULTS**

Default'lar bir sütuna değer girilmemesi halinde varsayılan olarak veritabanı tarafından yazılması, istenen değeri temsil eder. Yani default değere sahip bir sütuna kayıt girmeyebilirsiniz ve bu takdirde otomatik bir değer atama gerçekleşir.

### **USERS & ROLES**

User'lar SQL Server sistemine bağlanmak isteyen, erişim hakkı tanınmak istenen kullanıcılar için verilecek veritabanına erişim yetkilendirme nesnesidir.

Role ve User'lar birbirinin tamamlayıcısıdır. Bir User'ın SQL Server sisteminde hangi görevde, yetkiye, erişim hakkına sahip olduğunu Role'ler ile belirleyebiliriz. Bir User'a birden fazla Role atanabilir.

### **RULES**

Rule'ler Constraint'ler gibi veri sınırlama işlemlerinde kullanılır. Tabloya eklenecek ya da güncellenecek veri, kural dışı ise bu işlem reddedilir.

Rule'ler Microsoft tarafından SQL Server'ın sonraki versiyonlarında kaldırılacağı için bu nesneyi kullanmakta dikkatli olmanız ve büyük çaplı projelerde kullanmamanız önerilir.

Default için en çok kullanılan örneklerden biri veri girişi sırasında tarih, saat sütununa `GETDATE()` gibi bir otomatik 'o anki zaman' bilgisini veren değerin girilmesidir.

## TANIMLAYICILAR VE İSİMLENDİRME KURALLARI

SQL Server içerisinde her şey bir nesne ya da tanımlayıcıdır ve bunların programsal olarak kullanılabilmesi için tabi ki isimlendirilmiş olması gereklidir.

Aşağıdakiler isimlendirilen nesnelere örnek olarak verilebilir.

View	Login	Server
Stored Procedure	Index	Role
Trigger	User	Constraint
User-Defined Function	Database	Column
User-Defined Type	Table	File
Schema	File	Full-Text Catalog

Ve bu isimlendirilebilir nesnelere daha birçok örnek verilebilir.

## İSİMLENDİRME KURALLARI

SQL Server'da boşluklar ve anahtar kelimelerle aynı olan isimlendirmeler yapılabilir. İsimlendirme konusunda SQL Server esnektir. Ancak bazı kurallara uymak ve esnekliğin şartlarına uymak gereklidir.

Uyulması gereken ana isimlendirme kuralları şunlardır;

- Normal nesneler için 128 karakter, geçici nesneler için 116 karakter uzunlığında isimler verilebilir.
- Tanımlama için kullanılacak ismin ilk harfi Unicode 2.0'a uygun bir harf olmak zorundadır. Türkçe karakter kullanılmamalı, büyük ya da küçük harf olarak A-Z arasında Türkçe olmayan karakterler ile başlamalıdır. İlk harften sonra bu kural geçerli değildir ve farklı karakterler kullanılarak isimlendirme yapılabilir.

- SQL Server anahtar kelimeler ile aynı olan ya da boşluk içeren isimler, köşeli parantez ([ ]) ya da tırnak (" ") içerisinde gösterilmelidir. Bunun amacı, veritabanınızı kullanmak istediğiniz esnek kurallara göre bilgilendirmektir.

Çift tırnak özelliğini kullanabilmeniz için `QUOTED_IDENTIFIER` özelliği `SET QUOTED_IDENTIFIER ON` şeklinde `SET` etmeniz gerekmektedir.

Köşeli parantez kullanımı da [boşluklu tanımlayıcı] şeklinde boşluklar için kullanılabilir. Eğer boşluk kullanılacak ise bu şekilde köşeli parantez kullanılması önerilmektedir.

Bu kuralların bir kısmı genel olarak programlama kurallarıdır. Her platformda olduğu gibi yukarıda bahsettiğimiz kurallar da SQL Server platformu ve tüm tanımlamalar için geçerlidir.



Bu esnek kurallarının SQL Server mimarisinde var olması, kullanılmasını tavsiye niteliğinde algılanmamalıdır. Aksine bu esneklik kurallarını çok gereklilik duyulmadığı takdirde kullanılmamalıdır.



# T-SQL'E GENEL BAKIŞ

2

## TRANSACT-SQL KAVRAMI

**Transact-SQL (T-SQL)**, SQL Server'ın programsal alt yapısı için geliştirilen, ileri seviye veritabanı geliştirme ve yönetim özellikleri bulunan bir **sorgulama dilidir**.

T-SQL ile ilgili gerekli detaylı eğitim ve anlatımlar ileriki bölümlerde yapılmıştır.

Bu bölümde T-SQL ile ilgili temel ve en çok kullanılacak özellikler tüm detaylarıyla inceleyeceğiz.

Bu bölümde öğreneceğimiz T-SQL cümleleri şunlardır;

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**

Bu dört sorgu ifadesi T-SQL sorgu dilinin en temel sorgu özellikleridir. Veri İşleme Dili (**DML = Data Manuplation Language**) olarak bilinen bu sorgu ifadeleri, veritabanında veri seçme, veri ekleme, veri güncelleme, veri silme gibi temel işlemleri gerçekleştirir. Bir veritabanı uygulamasında en çok kullanılan ve uygulamanın **veritabanı uygulaması** olmasını sağlayan T-SQL ifadeleridir.

T-SQL sorgulama dili, SQL Server'ın tanıdığı ve sorgulamaları kendi veritabanı motorunda diğer RDBMS veritabanı motorları tarafından tanınamayacak bir dildir. Ancak bununla birlikte T-SQL'in SQL standartlarını da desteklediğini belirtmiştik. SQL Server giriş seviye ANSI SQL-92 standartlarını destekler. ANSI standartlarındaki SQL çoğu durumda T-SQL'e göre daha performanslı olacaktır. Eğer geliştirdiğiniz veritabanı uygulamasının her RDBMS için geçerli ve performanslı olmasını isterseniz, kullanmanız gereken sorgu dili T-SQL değil, SQL standartları olmalıdır. Yani, performans ve RDBMS platformlarından bağımsız bir veritabanı oluşturmak için mümkün olduğunda ANSI standartlarına uyumlu olmalısınız.

## **T-SQL İLE İLGİLİ KURALLAR**

### **NESNE VE DEĞİŞKEN İSİMLENDİRME KURALLARI**

T-SQL'de tablo, sütun, index vb. diğer tüm nesnelerin isimleri belirlerken uymamız gereken bazı standartlar var. Bu standartların sebebi diğer programlama dillerinde olduğu gibi çeşitli hatalara karşı uygulamamızı korumak ve programlamayı daha düzenli ve geliştiriciler için anlaşılabılır şekilde hazırlamak.

- Harf ([a-z] ya da [A-Z]) ile ya da alt çizgi (\_) ile başlamlıdır.
- Tanımlayıcı isimleri @, @@, #, ##, \$ gibi özel karakterler ile başlamamalıdır.

Bu özel karakterler sistem tarafından şu amaçlar için ayrılmıştır;

- @ : Değişken tanımlarken değişken adının başında kullanılır.
- @@ : Sistem değişkenlerinin isimlerinin başına gelir.
- # ve ## : Geçici nesne belirteci olarak kullanılır.
- SQL ve T-SQL mimarisinde tanımlanmış (SELECT, INSERT, UPDATE gibi) kelimeler kullanılmamalıdır.
- Tanımlayıcı isimlerinde Türkçe harfler (Ü, ü, Ç, ç, Ö, ö, Į, į, Ĝ, ģ, Š, š) ve boşluk yer almamalıdır.

Bu kurallara uymayan bir tanımlayıcı kullanılacak ise tanımlayıcı isimleri köşeli parantez ([]) ya da (‘’) çift tırnak içerisinde yazılması gereklidir.

Çift tırnak kullanımı için gerekli açıklama **SQL Server Veritabanı Nesneleri** bölümündeki **İsimlendirme Kuralları** konusunda anlatılmıştır.

## TANIMLAYICI İSIMLENDİRME NOTASYONLARI

### NOTASYON KAVRAMI

Tüm programlama dillerinde değişken ve nesne isimlendirmelerinde bazı kısıtlamalar mevcuttur. Tavsiye edilenlerden bazıları boşluk bırakmamak ve çoğu karakterin kullanılamamasıdır. Bu durumda otomatik olarak isimlendirilen nesnelerin anlaşılması konusunda da bazı kısıtlamalar getirilmiş oluyor.

Bu kısıtlamalar, notasyon kavramı ile bir nebzede dengelenmiştir. Notasyonlar tanımlama işlemlerinde bazı kurallar ve düzen getirmeyi hedefler. Tanımlama işlemini büyük ya da küçük harf ile başlatmak, kelime arasına alt çizgi koymak gibi sıralanabilir. Birçok farklı notasyondan kabul gören bazıları aşağıdaki gibi biridir.

### CAMEL NOTASYONU

İlk kelimenin tamamen küçük harf, geri kalan tüm kelimelerin baş harflerinin büyük şekilde yazılmıştır.

**Örnek:** `ileriSeviyeSqlServerTsql, kodLab, dijibilCom`

### PASCAL NOTASYONU (DEVE NOTASYONU)

Kelimelerin ilk harflerinin büyük diğer harflerin küçük yazılmasıdır.

**Örnek:** `İleriSeviyeSqlServerTsql, KodLab, DijibilCom`

### ALT ÇİZGİ (UNDERSCORE) NOTASYONU

Kelimelerin arasına altçizgi koyulmasıdır.

**Örnek:** `İleri_Seviye_SQL_Server_Tsql, Kod_Lab, Dijibil_Com`

### BÜYÜK HARF (UPPERCASE) NOTASYONU

İsimlendirmenin büyük harflerle yapılmasıdır. Tüm notasyonlarda olduğu gibi anlamayı kolaylaştırmak için bu notasyon ile diğer bir notasyon olan alt çizgi kullanımını gerçekleştirilebilir.

**Örnek:** `İLERİSEVİYESQLSERVERTSQL, KODLAB, DIJIBILCOM`

**Alt Çizgili:** `İLERİ_SEVIYE_SQL_SERVER_TS SQL, KOD _LAB, DIJIBIL _COM`

## MACAR NOTASYONU

Pascal notasyonuna ek olarak isimlendirmenin başına, kullanılan değişken tipinin veya kontrolün kısaltma ile yazılmasıdır.

**Örnek:** `tblKODLAB, spDIJIBILCOM (sp = stored procedure, tbl = tablo)`

## AÇIKLAMA SATIRLARI

Tüm programlama dilleri ve veritabanı yazılımlarında var olan ve geliştiricilerin kod blokları arasında açıklamalar, bilgi ve hatırlatıcı metinler girebilmesini sağlayan bir özelliklektir. Bu açıklama satırı, tek satır ve çok satırlı olabilmektedir.

Açıklama satırı için iki farklı yöntem vardır. Bunlar iki tane yan yana tire işaretti (--) ve bu şekilde /\* \*/) kullanılmıştır.

Açıklamanız tek satır ise;

```
-- 'CREATE TABLE kodlab' komutu ile kodlab adında bir tablo oluşturun.
```

Açıklamanız çok satırlı ise;

```
/*
 'CREATE TABLE dijibil' komutu ile
 dijibil adında bir tablo oluşturun.
*/
```

Açıklama haline getirilen satırlar otomatik olarak açık yeşil ile renklendirilir ve kodların veritabanı motoruna gönderilmesi durumunda veritabanı motoru buradaki metinleri kod olarak görmediği için çalışılmaz.

## NULL KAVRAMI

Bilgisayar teriminde boşluk ile hiçlik (**NULL**) aynı değildir. Veritabanında ilgili bir kayıt için klavyeden boşluk (**space**) tuşu kullanılarak değer verilebilir. Bu o kaydın olmadığı anlamına gelmez. Bu boş kaydın karşılığı ASCII’de (ASCII-32) vardır. Ancak boşluk dahil hiç bir değer girilmemiş, işlem yapılmamış kayıtlar **NULL** olarak nitelendirilir. ANSI Standardında değeri bilinmediği için **NULL** değerler, **NULL** değer kontrolü yani kaydın **NULL** olup olmadığı hariç, herhangi bir işleme tabi tutulamazlar. İki **NULL** değerin birbirine eşit olup olmadığını kontrol etmek gibi bir işlem yapılamaz.

## T-SQL'DE YIĞIN KAVRAMI

SQL Server'da yiğin kavramı, sorguların sırayla işleme alınmasını ifade eder. Çalışma sırasında SQL Server'a birden fazla gönderilen sorgu yiğinlar halinde ele alınır.

### GO KOMUTU

GO komutu bir yiğinin sonunu belirtmek için kullanılır. Bir yiğin SQL Server'da işlenmeye başladığı anda önce ayrıştırılır (*parse*), sonra derlenir (*compile*) ve son olarak sorgu çalıştırılır (*execute*).

#### Kullanımı:

```
Komutlar
GO
```

### USE KOMUTU

T-SQL kod bloklarının, hangi veritabanı için çalıştırılacağını belirtmek için kullanılır.

#### Kullanımı:

```
USE
Veritabani_Adı
```

### PRINT KOMUTU

T-SQL içerisinde kullanılan değişkenlerin değerleri ya da hatalar sonucu oluşacak hata mesajları gibi sonuçların sorgu çalıştığı anda gösterilmesini sağlar.

#### Kullanımı:

```
PRINT gosterilecek_deger
```

#### Örnek 1:

```
PRINT 'print edilecek değer' -- Sıradan bir metinsel ifade de olabilir.
```

#### Örnek 2 :

```
PRINT @degisken_adi
```

**Örnek 3:**

```
USE kodlab
GO
DECLARE @kitap VARCHAR(29)
SET @kitap = 'İleri Seviye SQL Server T-SQL'
GO
PRINT @kitap
GO
```

Kitap değişkenine, boşluklar dahil 29 karakterlik veri girildiği için **VARCHAR (29)** veri boyutu atanmıştır.

**VERİ TANIMLAMA DİLİ**

Veri Tanımlama Dili (**DDL = Data Definition Language**), SQL Server'da veritabanı nesneleri üzerinde yönetim işlemleri gerçekleştirmek için kullanılır. **CREATE** ile veritabanı, tablo, index, trigger, stored procedure ya da bir başka veritabanı nesnesi oluşturabilir. **ALTER** ile daha önce oluşturulmuş bir nesne üzerinde yapısal değiştirme işlemi gerçekleştirilir. **DROP** ile de mevcut bir veritabanı nesnesi silinir.

Kısaca özetlemek gerekirse;

- CREATE** Herhangi bir veritabanı nesnesi oluşturur.
- ALTER** Mevcut bir nesneyi yapısal olarak değiştirir.
- DROP** Mevcut bir nesneyi veritabanından siler.

**SQL SERVER'DA NESNE İSİMLERİ**

SQL Server Management Studio'da sorgu hazırlarken görünen araçları kullanarak giriş sırasında server seçmek ve bağlanmak, sorgu ekranında veritabanı seçimi gibi kolaylıklarından dolayı önemli bir nokta olan nesne isimlendirmelerini özellikle incelememiz mimariyi anlamak adına faydalı olacaktır.

SQL Server, bize kolay kullanım ve çeşitli araçlar sunar. Fakat arka planda aslında her şey nesneler şeklinde birbirine bağlı olarak seçilir ve ilişkilendirilir. Soldaki Tree (ağaç) yapısından bir veritabanı seçersiniz, sonra bu veritabanı içerisinde tablolar sekmesini açar ve işlem yapacağınız tabloyu seçerek istediğiniz işlemi yaparsınız.

Programsal olarak bu yapı veritabanında şu şekilde ilişkilendirilerek yönetilir.

---

```
[server_ismi].[veritabani_ismi].[sema_ismi]].nesne_ismi
```

---

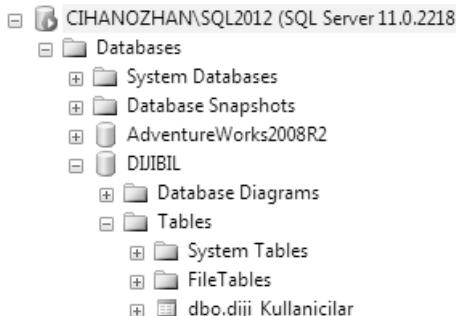
Bir **sorgu ekranı** (*Query Window*) açtıktan sonra, üst kısımdaki **Available Databases** kısmında işlem yapacağınız veritabanı seçili ise, sadece nesne adını belirterek işlem yapabilirsiniz. Bu ilişkilendirme olayı SSMS ile şu şekilde gerçekleştirilebilir.

- SSMS açılırken **Server Name** kısmında bağlanacağınız server'ı seçerek **Connect** butonu ile bağlanılır.
- Sol **Tree** yapısında bağlanılan server içerisindeki **Databases** sekmesinden veritabanı sekmesi genişletilir.
- Seçilen veritabanı içerisinde **Tables** bölümünden işlem yapacağınız tabloyu seçebilirsiniz.

---

```
CIHANOZHAN\SQL2012.DIJIBIL.dbo.diji_Kullanicilar;
```

---



Bu sıralamanın doğruluğundan emin olmak için kendi server, veritabanı ve nesne adınızı göre aşağıdaki gibi sorgu hazırlayıp çalıştırabilirsiniz.

---

```
SELECT * FROM [CIHANOZHAN\SQL2012].DIJIBIL.dbo.diji_Kullanicilar;
```

---

Sorgumuzda **server\_ismi** değerim olan **CIHANOZHAN\SQL2012** değerini kare parantez (**[]**) içerisine almamın sebebi; sorgunun veritabanı motoruna doğru şekilde gönderilmesini ve yazım da kullanılan karakterin hata vermeden çalışmasını sağlamak içindir. Kare parantez ile ilgili detaylı açıklamaları, nesne isimlendirme kuralları konusunda yapmıştık.

## **SCHEMA İSMİ (OWNERSHIP / SAHİPLİK)**

Schema (*şema*), ANSI standartlarında var olan bir standarttır. SQL Server'da schema özelliği eski sürümlerden beri desteklenmektedir. Fakat bu isimlendirme ownership, owner gibi farklı şekillerdeydi. SQL Server sahiplik anlamına gelen ve aynı hizmet için var olan bu özelliğin adını ANSI standartlarına uyumlu hale getirerek **schema** adını vermiştir. Schema kavramı bir ANSI standardı olmakla birlikte, sadece SQL Server'da değil, Oracle gibi diğer veritabanlarında da bu şekilde ve bu amaç için kullanılır.

Önceki sürümlerde SQL Server'da, sahip nesneyi oluşturan ya da veritabanı sahibiydi. Fakat yeni versiyonlarla birlikte nesne bir sahipten çok bir schema'ya atanır. Sahiplik belirli bir kullanıcıya bağlı iken, schema birden fazla kullanıcı tarafından paylaşılabilir ya da bir kullanıcı birden fazla schema hakkında sahip olabilir.

### **VARSAYILAN SCHEMA: dbo**

SQL Server'da veritabanını oluşturan kullanıcı, veritabanının sahibi ya da dbo olarak tanımlanır. Veritabanı sahibi olan kullanıcı tarafından oluşturulan bir nesne, kullanıcının kullanıcı ismi schema'sında değil, dbo schema içinde listelenecektir.

Veritabanı sahibi olmayan bir kullanıcı için şu şekilde örneklendirilebilir.

KodLab adında bir veritabanımız olduğunu ve bu veritabanında veritabanı sahibi olmayan, Cihan adında bir kullanıcı olduğumu düşünelim. Bu veritabanında **CREATE TABLE**, yani tablo oluşturma yetkisine sahibim. KodLab veritabanında **Kitaplar** adında bir tablo oluşturursam, bu tablonun sahibiyle isimlendirilmiş nesne ismi **Cihan.Kitaplar** şeklinde olacaktır. Bu kullanım ile Cihan kullanıcısı, KodLab veritabanındaki **Kitaplar** tablosunun özel bir sahibi olduğundan, bir başka kullanıcı bu tabloya erişmek istediginde schema ismi farklı olacağı için **Cihan.Kitaplar** şeklinde sahibi nitelendirilmiş nesne ismini vermesi gereklidir. Bu durumda sahibi nitelendirilmiş nesne ismimiz **Cihan.Kitaplar** olacaktır. Eğer bu tabloyu veritabanı sahibi olan bir kullanıcı oluşturmuş olsaydı bu tablo, varsayılan şema olan dbo şeması içerisinde oluşturulacağı için schema belirtmeye gerek kalmayacak ve **Kitaplar** şeklinde nesneye ulaşabilecekti. Veritabanı sahibi tarafından oluşturulan nesnelere **dbo.nesne\_ismi** ya da sadece **nesne\_ismi** şeklinde ulaşılabilir.

**sa** (*system administrator*) ya da **sysadmin** rolü üyelerini daima dbo takma adını kullanır. **sa**'lar aslında veritabanının gerçek sahibi değildir. Fakat yetki olarak veritabanı sahibinin (dbo) yetkilerine sahiptirler ve **sa** kullanıcılarının oluşturacağı nesnelerin sahibi dbo olacaktır. **db\_owner** veritabanı rolüne ait kullanıcılar tarafından oluşturulan nesneler için ise; varsayılan schema dbo değil, kullanıcının varsayılan schema'sı olacaktır.

## VERİTABANI İSMİ

Veritabanında nesne isimlendirmelerinden bahsederken **schema\_ismi** bilgisinden sonra **veritabani\_ismi** bilgisini belirmemiz gerektiğini öğrenmiştık. Veritabanı ismi nesne isimlendirmeleri ve ilişkilendirmelerinde önemli bir yere sahiptir.

Bir tane **DIJIBİL**, bir de **KodLab** adında veritabanımız olsun. Biz **DIJIBİL** veritabanında işlem yapmak için oturum açtığımızı ve işlem sırasında **KodLab** veritabanındaki Kitaplar tablosundan **DIJIBİL** veritabanındaki **diji\_Kitaplar** tablosu arasında **JOIN** ile ilişkilendirme yapacağımızı ya da sıradan bir **SELECT** ile diğer veritabanına geçmeden diğer veritabanındaki bir tabloyu sorgulamak ya da herhangi bir işlem ile iki ayrı veritabanındaki iki farklı tablo arasında işlem yapacağımızı varsayıyalım. Bu tablolar arasında işlem yapabilmemiz için veritabanlarının isimlerine ihtiyacımız vardır.

**KodLab** veritabanı ile oturum açtığımızda, **DIJIBİL** veritabanına ait bir tabloya sorgulama yapmak için;

---

```
SELECT * FROM DIJIBIL.dbo.diji_Kullanicilar;
```

---

**dbo** varsayılan bir schema olduğu için **dbo** yazmadan da şu şekilde çalışılabilir;

---

```
SELECT * FROM DIJIBIL..diji_Kullanicilar;
```

---

**SSMS** ile oturum açığınız ve sorgu penceresi (Query Window) oluşturduğunuzda size geçerli veritabanına sorgu yapmaya hazır bir ekran gelir. Bu kısımda gerçekleştirdiğiniz sorgular her zaman geçerli veritabanında sorgulanır. Bu geçerli veritabanı sizin için varsayılan veritabanıdır.

SQL Server'da dbo varsayılan schema'dır. Bir kullanıcı, oluşturduğu nesne için schema belirtmemişse bu nesne SQL Server tarafından ilk olarak dbo nesnesinde aranır. Bu nedenle, dbo içerisinde olmayan bir nesne belirttiğinize eminseniz performans açısından SQL Server'ın dbo içerisindeki nesneleri araması sırasındaki kaybı önlemek için schema adını belirtmenizde fayda vardır.

## **SERVER TARAFINDAN İSİMLENDİRME**

SQL Server, içerisinde sadece veritabanları arasında bağlanma ve sorgular geliştirmeye değil, aynı zamanda farklı server'lar arasında da sorgular geliştirmeye izin verir. Bu server'lar sadece SQL Server değil, Oracle, DB2 ya da benzeri büyük veritabanlarını da destekler.

Bağlantılı server'lar (*Linked Servers*), geliştiriciden çok veritabanı yöneticilerinin görev alanına girer. Bu nedenle bu özelliğin nasıl isimlendirildiğini öğrenerek devam edeceğiz.

Bir server ile bir başka server arasındaki bağlantı ile isimlendirme şu şekilde gerçekleşir;

---

```
sunucu_ismi.veritabani_ismi.sema_ismi.nesne_ismi
```

---

## **CREATE İLE NESNE OLUŞTURMAK**

**CREATE** ifadesi ile veritabanındaki nesnelerden herhangi biri oluşturulabilir. **CREATE** ifadesi genel bir oluşturucu ifadedir. **CREATE** ifadesinden sonra oluşturulacak nesnenin özelliklerine göre farklı parametreler alır.

---

```
CREATE nesne_adi
```

---

### **CREATE DATABASE İLE VERİTABANI OLUŞTURMAK**

SQL Server'da bir veritabanı oluşturmanın en temel hali şu şekildedir;

---

```
CREATE DATABASE veritabani_adi
```

---

Bize şablon görevi gören model veritabanımızdaki varsayılan ayarlar ile bir veritabanı oluşturmak bu kadar kolaydır. Ancak SQL Server'da tüm detaylarıyla

hakim olmanızı önerdiğim konulardan biri de, T-SQL kullanarak veritabanı oluşturmak ve bu kullanım sırasındaki karışık bir çok parametrenin ne anlam ifade ettiğini öğrenmenizdir. İleri seviye veritabanı programlama yapmak için veritabanını oluşturan T-SQL yapısını iyi kavramanız gereklidir. Bir veritabanı için varsayılan ayarların ne olduğunu **SQL Server Veritabanı Nesneleri** bölümündeki `model` konusunda anlatmıştık.

SQL Server'da yeni bir nesne oluşturulduğunda, bu nesneyi oluşturan kişi sistem yöneticisi ya da veritabanı sahibi değil ise, nesneyi oluşturan kişi haricinde kimse bu nesneye erişemez. Tabi ki bu oluşturulan nesnenin gerekli erişim ayarlarını daha sonra da ayarlayarak değiştirebilirsiniz.



T-SQL ile veritabanı oluşturmak için Microsoft tarafından hazırlanan söz dizimi ve kuralları içeren script çok uzun olduğu için burada yayınlamıyorum. Ancak dilerseniz aşağıdaki bağlantıdan ulaşabilirsiniz.

<http://msdn.microsoft.com/en-us/library/ms176061.aspx>

Yukarıda verdığım script dosyası ya da bağlantıdan edindiğiniz kodların önemli olan bazı özellikleri tek tek detaylarıyla inceleyelim.

## ON

Veritabanının veri ve log dosyasının yerini belirtmek için kullanılır. `ON` operatöründen sonra gelen `PRIMARY` anahtar kelimesini fark etmiş olmalısınız. Bunun anlamı, belirtilen dosya bilgilerinin fiziksel olarak birincil dosya grubuna ait olduğunu.

## NAME

Veritabanının mantıksal ismini belirtir. SQL Server veritabanı dosyasına ulaşmak için bu ismi kullanır. Veritabanını ya da dosyayı yeniden boyutlandırmak gibi işlemler için bu isim kullanılır.

## FILENAME

Veri ve log dosyasının bulunduğu dosyaların işletim sistemi üzerindeki gerçek fiziksel dosya isimlerini belirtir. İsimlendirme veri ve log dosyası için farklı şekilde yapılır.

Veri dosyası uzantısı: **mdf**

İkincil dosya uzantısı: **ndf**

Log dosyası uzantısı: **ldf**

Veri ve log dosyası için varsayılan dosya yolu;

*Program Files(x86)\Microsoft SQL Server\MSSQL10\_50.MSSQLSERVER\MSSQL\DATA*

Bu dosya yolundaki **DATA** içerisinde **DİJİBİL** adındaki bir veritabanının veri ve log dosyaları varsayılan olarak şu şekilde isimlendirilir;

Veri Dosyası: **DIJIBIL.mdf**

Log Dosyası: **DIJIBIL\_log.ldf**

Bilgisayarınız 64 Bit destekli ise, kurulumunu yaptığınız SQL Server'a göre Program Files (x86) klasörü içerisinde bulunabilir. Örneğin, benim bilgisayarımda SQL Server 2008 ve SQL Server 2012 ayrı ayrı kuruludur.

SQL Server 2012'nin kurulu olduğu dosya yolum ise;

*Program Files\Microsoft SQL Server\MSSQL11.SQl2012*

SQl2012 ismi, iki veritabanı Instance'ının farklı olması gerektiği için kurulum sırasında verdiği bir isimdir.

## VERİ TERİMLERİ BÜYÜKLÜKLERİ

SQL Server, veriyi tüm bilgisayar sistemlerindeki standartlara göre depoladığı ve yönettiği için, veritabanı boyutlandırma ya da benzeri boyut hesaplamalarında aşağıdaki eşleştirmelere uymanız gereklidir.

**1 Bit(b)**

**1 Byte(B)** = 8 bit

**1 KiloByte(KB)** = 1024 Byte

**1 MegaByte(MB)** = 1024 KiloByte

**1 GigaByte(GB)** = 1024 MegaByte

**1 TeraByte(TB)** = 1024 GigaByte

**1 PetaByte(PB)** = 1024 TeraByte

**1 ExaByte(EB)** = 1024 PetaByte

**1 ZettaByte(ZB)** = 1024 ExaByte

**1 YottaByte(YB)** = 1024 ZettaByte

Zettabyte Dünyada ki bütün verilerin toplamının ancak 1,8 Zettabyte olduğunun tahmin edildiğini ve 1 Zettabyte'ın 250 milyar adet DVD ya da 36 milyon yıllık HD video görüntüsüne denk geldiğini düşünürsek, şimdilik YottaByte'ı bilmesek de olur

## SIZE

Veritabanının dosya boyutunu gösterir. Boyut varsayılan olarak **MB (MegaByte)** cinsinden belirtilir. Ancak boyutu **KB (KiloByte)**, **GB (GigaByte)** ya da **TB (TeraByte)** cinsinden de belirtebilirsiniz.

**model** veritabanını anlatırken bahsettiğimiz gibi, dosya boyutu en az model veritabanı boyutu kadar olması gereklidir. Dosya boyutu belirtilirken tam sayı cinsinden, yani ondalık bölüm olmadan belirtilmesi gereklidir.

**SIZE** parametresiyle bir değer belirtmezseniz varsayılan olarak model veritabanı boyutu ile oluşturulacaktır.

## MAXSIZE

Veritabanı dosya boyutunun genişleyebilecegi en yüksek büyülügü belirtir. **MAXSIZE** parametresi de **SIZE** gibi varsayılan olarak MB cinsinden değer alır. Fakat dilerseniz KB, GB ya da TB cinsinden dosya boyutu belirtebilirsiniz. Bu parametre değerinin belirlenmesi zorunlu değildir. Eğer bir değer belirtilmezse varsayılan olarak bir **MAXSIZE** değeri atanmayacak ve bu durumda **MAXSIZE** değeri sabit disk (HDD)'in kapasitesiyle sınırlı olacaktır.

**MAXSIZE**, üzerinde sürekli anlık işlem yapıldığı durumlarda çok kritik ve takip edilmesi gereken bir özelliklektir. Eğer dosya boyutları (mdf, ndf), **MAXSIZE**'in maksimum değerine ulaşırsa, kullanıcılar veri girişi yapmak istediğiinde hata alırlar. Log tutma işleminde de maksimum değere ulaşıldığında log tutulamayacaktır. Bir veritabanı veri dosyası (mdf), sadece veri giriş işlemlerinde bir girdi olduğu için hızlı büyümeyecektir. Ancak log dosyaları her işlem için log dosyasına (ldf) loglama işlemi gerçekleştirdiği için sürekli büyüyecektir.



Bir veritabanının **MAXSIZE**'ı verilmediği takdirde üst limit sabit diskin üst sınırı ile eş değerdir. Veritabanı sürekli genişleyecek ve sabit disk doldurulana kadar herhangi bir sorun çıkarmayacaktır. Ancak sabit disk alanını da doldurduğunda, bu sadece bir veritabanı hatası değil, işletim sistemi dahil diğer tüm programların da çalışmamasına sebep olacaktır. Bu nedenle size tavsiyem, mutlaka bir **MAXSIZE** değeri ve **FILEGROWTH** değeri belirlemeniz yönünde olacaktır. Kontrolü SQL Server'a bırakmayın. Tüm ayar ve yönetimi kendiniz belirleyin.

## FILEGROWTH

**FILEGROWTH** özelliği, veritabanı dosyasının bir seferde ne kadar genişleyeceğini(büyüyeceğini) belirtmek için kullanılır. Veritabanının başlangıç genişlik değeri **SIZE** ile olabilecek en yüksek değeri **MAXSIZE** arasında geçerlidir. Bu değeri KB, MB, GB ya da TB olarak belirleyebilirsiniz. Dilerseniz bu özelliğin yüzdesel olarak da belirleyebilirsiniz. Başlangıç değeri 100 MB olan bir veritabanının her seferinde %10 büyümeyi isterseniz, 10 MB büyüterek 110 MB olacaktır. **MAXSIZE** sınırını ve sabit disk (HDD) üst sınırınızı da hesaba katarak, bir büyümeye oranı belirlemeniz gereklidir.

## LOG ON

Log bilgilerinin tutulacağı özel dosyalar kümesi ve bu dosyaların bulunacağı yeri belirlemenizi sağlar. Bu özellik belirtilmezse, log dosyası (ldf) boyutu, veri dosyası (mdf) boyutunun %25'i olacak şekilde tek bir dosya oluşturur.

Profesyonel uygulamalarda veri güvenliği ve sabit disk yazma-okuma gibi bir çok parametre hesaplanarak veritabanı mimarisi oluşturulur. Log dosyası ile veri dosyasının konumlarını belirlemek için de dikkat edilmesi gereken, bu iki dosyanın farklı dosyalarda saklanmasıdır. Hatta log ve veri dosyasının farklı sabit disklerde tutulmasını tavsiye ederim. Hem veritabanı ve sabit disklerin veri okuma-yazma hızını daha etkin kullanabilmek hem de sabit diskte oluşabilecek çökme, disk sürücü hataları gibi durumlarda ek güvenlik sağlayacaktır.

```
LOG ON
( NAME = kodlab_db_log,
  FILENAME = 'C:\Databases\kodlab_db\kodlab_db_log.ldf',
  SIZE = 5MB,
  MAXSIZE = 25MB,
  FILEGROWTH = 5MB );
```

## COLLATE

Veritabanında büyük/küçük harf duyarlılığı, işaret duyarlılığı ve sıralama düzeni gibi özellikleri belirler. SQL Server kurulumu sırasında varsayılan olarak belirlenir. Ancak veritabanı seviyesinde bu özellikleri değiştirebilirsiniz.

## **FOR ATTACH**

Mevcut veritabanı dosyaları kümesini, server'a bağlamak için kullanılır. Bu dosyalar, daha önce veritabanına bağlı olan ve `sp_detach_db` komutu ile veritabanından uygun şekilde ayrılmış dosyalardır. Veritabanına dosya eklemek için `FOR ATTACH` kullanmak zorunda değilsiniz. `sp_attach_db` komutunu kullanarak da dosyaları veritabanına ekleyebilirsiniz. Fakat `FOR ATTACH` özelliği kullanılarak oluşturulan bir `CREATE DATABASE` sorgusunda 32.000 adet dosyayı bağlayabiliriz. Bu dosya bağlayabilme sayısı `sp_attach_db`'de sadece 16'dır.

`FOR ATTACH` özelliğini kullanacaksanız, dosya konumu bilgisinin `ON PRIMARY` kısmını tam olarak belirtmelisiniz. Veritabanından ayrılmış dosyaları, aynı veritabanı dosyasına bağlayacağınız sürece, dosyanın konumunu gösteren diğer parametreleri kullanmayabilirsiniz.

## **DB\_CHAINING ON | OFF**

SQL Server'da **sahiplik** kavramı vardır. Bu özellik bir nesnenin sahibini ve bu nesneye erişim izinlerini yönetmek için kullanılır. Ancak sahiplik kavramı beraberinde sahiplik zinciri diye nitelendirdiğimiz bir sorunu getirir. Sahiplik zinciri, bir kişinin oluşturduğu nesneye bağlı başka bir nesnenin başka bir kullanıcı tarafından oluşturulmasıyla oluşur. Yani, A kişinin oluşturduğu nesneye bağlı bir nesnenin B kişi tarafından oluşturulması anlamına gelmektedir.

`DB_CHAINING ON` olarak belirlerseniz, veritabanı arasında sahiplik zincirleri çalışır. `DB_CHAINING OFF` olarak belirlerseniz çalışmaz.

Sahiplik zinciri nesnelere erişim, yönetim ve sahipliklerin içinden çıkalamaz haline gelebilmesini sağlayabilir. Bu nedenle bu tür sahiplik zinciri oluşturulabilecek durumlardan mutlaka kaçınmanızı tavsiye ederim.

## **TRUSTWORTHY**

**TRUSTWORTHY** özelliği, veritabanınızdaki bir nesnenin, sistem dosyası gibi SQL Server ortamı dışındaki nesnelere erişmek istedığınızı ve nesneye **impersonation context** erişimi vermek istedığınızı belirtir. Bu özelliği daha sonra detaylarıyla inceleyeceğiz.

## T-SQL İLE VERİTABANI OLUŞTURMAK

En temel haliyle, **KodLab** adında bir veritabanı oluşturalım.

---

```
USE master
CREATE DATABASE KodLab
```

---

Şimdi de **DIJIBİL** adında detaylı bir veritabanı oluşturalım.

İlk olarak, sabit diskimizin c:\ dizininde, veritabanı dosyalarımızın tutulacağı **Databases** adında bir klasör oluşturuyoruz. Bu klasörü oluşturmazsanız aşağıdaki sorgu çalıştırılma sırasında hata verecektir.

Daha sonra **SSMS**'de sorgu penceresi (*Query Window*) oluşturarak aşağıdaki kodları çalıştırıyoruz.

---

```
CREATE DATABASE DIJIBIL
ON PRIMARY
(
    NAME = N'DIJIBIL_Data',
    FILENAME = N'C:\Databases\DIJIBIL_Data.mdf',
    SIZE = 21632KB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 16384KB
)
LOG ON
(
    NAME = N'DIJIBIL_Log',
    FILENAME = N'C:\Databases\DIJIBIL_Log.ldf',
    SIZE = 2048KB,
    MAXSIZE = 2048GB,
    FILEGROWTH = 16384KB
)
GO
```

---

Bu sorgu ile birlikte artık **DIJIBİL** adında bir veritabanına sahibiz. Şimdi **Databases** klasörümüzdeki veritabanı dosyalarımızın isim ve boyut değerlerine bakalım. Sonra da bu sorguda neler yaptığımızı sırasıyla inceleyelim.

 DIJIBIL_Data.mdf	21.632 KB
 DIJIBIL_Log.ldf	2.048 KB

Veritabanı oluşturmak için gerekli olan en temel ifadeyi yazıyoruz.

---

```
CREATE DATABASE DIJIBIL
```

---

Bu veritabanına ait birincil dosya grubunun belirtileceğini bildirmemiz için gerekli ifadeyi yazıyoruz.

---

```
ON PRIMARY
```

---

Bundan sonraki bilgileri, iki ayrı parantez içerisindeki iki ayrı blokta belirtiyoruz.

---

```
(  
    NAME = N'DIJIBIL_Data',  
    FILENAME = N'C:\Databases\DIJIBIL_Data.mdf',  
    SIZE = 21632KB,  
    MAXSIZE = UNLIMITED,  
    FILEGROWTH = 16384KB  
)
```

---

Bu ilk kod bloğunda veritabanımızın veri dosyasını tanımlıyoruz.

- **NAME** : Veri dosyasının adı.
- **FILENAME** : Veri dosyamızın işletim sistemindeki fiziksel adı.
- **SIZE** : Veritabanı boyutunu 21632KB (KiloByte) olarak belirtiyoruz.
- **MAXSIZE** : Veritabanı boyutunu limitsiz olarak belirtiyoruz. Limit, sabit diskin kapasite üst sınırı olacaktır.
- **FILEGROWTH** : Veritabanı genişlemesi gerekiğinde, her seferinde 16384KB olarak artmasını istiyoruz.

Veritabanı için belirttiğimiz **KB** cinsinden değerleri **MB** cinsinden de verebilirdik. Bu durumda veri dosyamızın **SIZE** özelliğindeki **21632KB** değeri, **21MB (MegaByte)** olarak verebilirdik. (1MB = 1024KB)

Bu bilgiler veri dosyası için hazırlandı. Şimdi Log dosyamızın tanımlamasını inceleyelim.

Log dosyasını tanımlayacağımızı bildirmek için ilk parantezli kod bloğundan sonra;

---

```
LOG ON
```

---

Şimdi ikinci kod bloğu ile veritabanımızın log dosyasını belirtebiliriz.

---

```
(  
    NAME = N'DIJIBIL_Log',  
    FILENAME = N'C:\Databases\DIJIBIL_Log.ldf',  
    SIZE = 2048KB,  
    MAXSIZE = 2048GB,  
    FILEGROWTH = 16384KB  
)
```

---

Veri dosyası için belirttiğimiz tüm parametreleri log dosyası için de farklı değerler vererek kullandık.

Log dosyasındaki bilgilerin veri dosyasından farkları;

- **FILENAME** : Log dosyasının uzantısı **ldf**'dir. Ve veritabanı adından sonra Data değil, Log yazılır.
- **SIZE** : Başlangıç olarak log dosyası yüksek boyutlara sahip olmadığı için daha düşük verdik.
- **MAXSIZE** : Log dosyası sürekli genişleyeceği için belli bir boyut sınırı koyduk.

Temel ve detaylı olarak iki farklı veritabanı oluşturduk. Şimdi bu veritabanının yapısına bakalım.

**sp\_helpdb** sistem prosedürü ile bir veritabanının yapısını gözlemeylebilirsiniz. Bu sistem prosedürü, veritabanın içerisindeki nesneleri değil, yapısını incelemek için oluşturulmuştur.

---

```
EXEC sp_helpdb 'DIJIBIL';
```

---



**EXEC** ve **EXECUTE** komutları stored procedure'leri çalıştırmak için kullanılır. İlerleyen konularda detaylarıyla anlatılacaktır.

	name	db_size	owner	dbid	created	status	compatibility_level	
1	DIJIBIL	23.13 MB	Cihanozhan\Chan Özhan	9	Oct 19 2012	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	110	
	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	DIJIBIL_data	1	C:\Databases\DIJIBIL_Data.mdf	PRIMARY	21632 KB	Unlimited	16384 KB	data only
2	DIJIBIL_Log	2	C:\Databases\DIJIBIL_Log.ldf	NULL	2048 KB	2147483648 KB	16384 KB	log only

Bu sorgunun sonucunda gelen veriler ikiye ayrılır.

İlk veriler;

Veritabanı dosyasının mantıksal ve SQL Server mimarisindeki parametrik ayarlarını gösterir. Buradaki `db_size` ile veri ve log dosyasının toplam boyutunu, `dbid` değeri SQL Server'da kaç adet veritabanı oluşturulduğunu, veritabanın oluşturulma tarihi, sahiplik bilgisi (`owner`) gibi bilgileri gösterir.

İkinci veriler;

Veritabanının fiziksel dosyaları hakkında bazı bilgileri gösterir. Bunlar, ekran görüntüsünde gördüğünüz gibi anlaşılmasına kolay bilgilerdir.

**Video:** SSMS kullanarak bir veritabanı oluşturma detayları

## CREATE TABLE İLE TABLO OLUŞTURMAK

`CREATE` ile veritabanı oluşturabildiğimiz gibi tablo da oluşturabilmekteyiz. `CREATE` konusuna giriş kısmında söylediğimiz gibi, bu ifade tüm nesneleri oluşturmak için kullanır.

## TABLO VE SÜTUN İSİMLERİ

Tablo ve sütun isimlendirme bazı standartlara sahip olsa da genel olarak geliştiriciden geliştiriciye değişiklik gösterir. Bazı geliştiriciler bu standartları beğenmiyor olacak ki, okumanızın bile zor olduğu isimlendirmeler yapabilmektedir. Ancak size tavsiyem; mutlaka okunaklı, açık ve anlamı bozmayacak şekilde bir standart ile geliştirme yapmanızdır. Bu isimlendirmelerdeki sade ve okunaklılık sizin de işinizi kolaylaşacaktır.

Genel olarak nesne isimlendirme kuralları bu durumda da geçerlidir. Bu konuda gerekli açıklamayı **SQL Server Veritabanı Nesneleri** bölümünden inceleyebilirsiniz.

Diğer bazı kurallar şunlardır:

- İsimleri kısa ve anlamlı tutun.
- İsimlendirirken her kelimenin baş harfini büyük, devamında küçük harfler kullanın.

- Türkçe karakter (ı, ī, ğ gibi) kullanmayın.
- Boşluk bırakmamaya çalışın. Bitişik ve her kelime baş harfini büyük yazın. Boşluk mutlaka gerekliyse, köşeli parantez ( [ ] ) kullanın.
- Anlam bozulmayacak şekilde kısaltmalar kullanın (**No**, **ID**, **Net[Network]** gibi).

## VERİ TIPLERİ

Tablo tasarımlarında veri tipleri birçok sebeple önemlidir. SQL Server'da doğru veri tipini seçmek tecrübe ve veri tiplerine tam hakimiyet gerektirir. Doğru seçilmeyen veri tipi, performans kaybı, hata oluşturma, veri bütünlüğünü bozmak ve veritabanında gereksiz alan kaplama gibi bir çok soruna sebebiyet verebilir.

Bu nedenle uygulama mimarisinde kullanılması gereken veri tipini doğru seçmek için iyi araştırma ve proje analizi gerekmektedir.

## DEFAULT

Kullanıcı tarafından belirtilmeyen ya da belirtilmesi zorunlu olmayan durumlarda SQL Server tarafından verilen otomatik değerdir.

Buna bir örnek olarak, kayıt tarihini kullanıcından almak yerine, **GETDATE()** fonksiyonu ile o anın tarih ve zaman bilgisini, otomatik olarak sistem tarafından veritabanına girilmesi sağlanabilir.

Bununla ilgili kullanım ve anlatımları ileriki bölümlerde detaylarıyla yapacağız.

## IDENTITY

SQL Server'da **IDENTITY**, kayıt eklemeye gibi işlemlerde büyük öneme sahiptir. Bir sütunu, **IDENTITY** sütun olarak belirlediğinizde, SQL Server her kayıt için o sütuna sayı atar. Atanan sayı, **seed** (başlangıç) ve **increment** (artış miktarı) kadar artar ya da azalır.

**Seed** ve **increment** için varsayılan değer 1'dir. Bu, seed ile 1'den başlar, **increment** ile birer birer artar anlamına gelir. Bu varsayılan değerleri değiştirebilirsiniz. Örneğin, 5'den başlar ve beşer beşer artırılabilirsiniz. Bu durumda ilk kayıt için 5 değeri verilir ve 5, 10, 15 şeklinde artarak devam eder.

**IDENTITY** sütunlar sayısal olmak zorundadır. Otomatik artan özelliğinin sayısal olmaması zaten düşünülemez. Veri tipi olarak da **INT (integer)** ve

**BIGINT** kullanımı yaygındır. Ancak bunlardan daha küçük sayısal veri tipleri de kullanılabilir. Bu tamamen **IDENTITY** sütunun maksimum alabileceği değer ile alakalıdır. Çok yüksek kayıtlara ulaşmayacak bir sütun için **BIGINT** kullanmak İ/o'da gecikmeye ve performansı olumsuz yönde etkileyebilecek bir durum oluşturur.

**IDENTITY** sütun kullanırken, bazı durumlarda önceki kayıtlar için kullanılmış olan numaraları tekrar kullanmak isteyebilirsiniz. Örneğin; 1'den başlayıp 10'a kadar gidilmiş bir kayıtta 4. kayıt silinmişse, bu kayıt numarası boş kalacaktır. Boş kalan bu kayıt numarasını, yeni bir kayıt için kullanmak istersek, **SET IDENTITY INSERT ON** özelliğini kullanmalıyız. Bu özellik **IDENTITY** sütuna otomatik değer atanmasını engelleyecektir. Böylece **IDENTITY** değerli silinmiş kaydın yerine yeni bir kayıt ekleyebilirsiniz.

**IDENTITY** sütunlar, genel olarak tablolardaki birincil anahtarlarla yeni sayısal değer oluşturmak için kullanılır. **IDENTITY** sütun ile **PRIMARY KEY** (*birincil anahtar*) kavramı, bir arada çok sık kullanıldığı için aynı ya da benzer gibi görünse de tamamen farklı kavamlarıdır. **IDENTITY** belirli bir sayıdan başlayarak belirli bir aralık ile artan ya da azalan sayısal değer üretir. **PRIMARY KEY** ise, bir sütunda bulunan değerin benzersiz olmasını sağlar. **IDENTITY** bir değeri sıfırlayarak daha önce artırarak saydırığınız aralıkta yeniden saydırılmaya başlayabilirsiniz. Ancak sütunun **PRIMARY KEY** olması bu durumda ortaya çıkacak veri tekrarını önler. Yani **IDENTITY**, 1 ile 10 numaraları arasında birer birer artarak değer üretirken **PRIMARY KEY**'de bu üretilen değerlerin, o sütunda benzersiz olmasını garanti eder.

## **NOT FOR REPLICATION**

Replikasyon işlemi, veritabanının yerel ya da uzak sunucuya bazı bilgileri ya da tüm bilgileri kopyalanma işlemini gerçekleştirir. **NOT FOR REPLICATION** parametresi de veritabanı bir başka veritabanına kopyalandığında, bir kimlik değeri verilip verilmeyeceğini belirler.

## **ROWGUIDCOL**

Bu özelliği replikasyon işleminde bir **IDENTITY** sütun kullanmaya benzetebiliriz. İki tablo arasında veri ilişkilendirme yapıldığında **IDENTITY** sütun kısmı olarak yeterli gelebilir. Ancak daha karmaşık verilerin replikasyon işlemine tabi tutulduğu bir durumda **IDENTITY** bize doğru bir yetenek sunmayacağındır.

Bu durumda **IDENTITY** değerleri mutlaka çakışacak ve ileriye dönük bir çözüm sunamayacaktır.

Bu durumda, bize daha büyük veriler için unique (*benzersiz*) değer üretecek bir özelliğe ihtiyaç duyuyoruz. Bu tür sorumlara çözüm olması için oluşturulan **GUID** (*Global Unique IDentifier*) kavramı devreye girer. GUID 128-bit bir değer üreten, 38 basamaklı ondalık şeklinde bir değerdir. GUID ile her saniye yeni bir değer üretseniz bile, teorik olarak milyonlarca yıl sonra tekrar aynı değeri üretme ihtimaline sahiptir. GUID'in ürettiği değeri tekrar etme olasılığı, kimi hesaplamalara göre de milyarlarca değer de bir ihtimaldir. Büyük ihtimalle emin olmamız gereken şu ki; bu değer kolay kolay kendini tekrarlamayacaktır.

Bilgisayar ya da web programlamada da kullanılan GUID değerini üretmek için Windows işletim sisteminde gömülü **Win32 API** vardır. Bu API ile değer üretilerek kullanılır. Aynı şekilde bu değeri üretecek bir de **SQL Server fonksiyonu** vardır. SQL Server'da GUID değer üreten fonksiyonun adı **NEWID()**'dır.

Bu fonksiyonu kullanarak nasıl bir değer üretildiğine bakalım.

---

```
SELECT NEWID() AS GUID;
```

---

GUID	
1	E1E00357-10FA-4471-BBD9-E6D96B9907B3

## COLLATE

Sütun seviyesinde büyük/küçük harf duyarlılığı, işaret duyarlılığı ve sıralama düzeni gibi özellikleri belirler.

## NULL / NOT NULL

Tablodaki bir sütunun, **NULL** değeri kabul edip etmeyeceğini belirtmek için kullanılır. **NULL** olarak geçilemeyecek bir sütuna değer girilmez ve **NULL** bırakılırsa istenilen işlem yapılmayacaktır. **NULL** kavramı detaylı olarak uygulama konularında anlatılacaktır.

## SÜTUN KİSITLAMALARI

Tablodaki sütunuza girilecek veriyi kontrol edip bazı kısıtlamalar uygulamamız gerekebilir. Örneğin; tarih tutulacak bir alanda 100 sene öncenin

tarihinin girilmesine izin vermek, programsal bir mantık hatası olacaktır. Ya da yılın aylarını içeren bir sütunda 13. ay olarak bir değer girilmesine kısıtlama getirmemek bir hatadır.

Bu gibi sorunları önlemek için sütun bazlı kısıtlama yapılmalıdır.

## HESAPLANMIŞ SÜTUNLAR

Tablodaki sütunlar ile elde edilen, hesaplama ve anlık veri gösterimi gibi durumlarda görüntülenecek kayda bir sütun ismi vermemizi sağlayan özelliktir. Kendine ait bir değer ya da veri yoktur. Gösterdiği veriyi farklı işlemlerden alır. Amacı veriler için bir takma isim oluşturmaktır.

### Söz dizimi:

---

```
sutun_ismi AS hesaplanan_sutun_ifadesi
```

---

### Kullanımı:

---

```
SELECT FirstName + ' ' + LastName AS [İsim] FROM Person.Person;
```

---

## TABLO KISITLAMALARI

Tablo kısıtlamaları (*constraints*), tabloya eklenecek veri için kısıtlama getirmekte kullanılır. Tablo seviyeli kısıtlamalar **PRIMARY**, **FOREIGN KEY** ve **CHECK** constraint'leri içerir.

### ON

**ON** ifadesi, tablo tanımının hangi dosya grubu üzerinde yer alacağını gösterir. Bir tabloyu **ON** ifadesini kullanmayarak varsayılan dosya grubuna (**PRIMARY**) yerleştirebilirsiniz.

## TABLO OLUSTURMAK

Oluşturduğumuz DIJIBIL veritabanında **diji\_Kullanicilar** adında bir tablo oluşturalım.

---

```
USE DIJIBIL
GO
CREATE TABLE diji_Kullanicilar(
kul_ID      INT           NOT NULL,
```

```

kul_ad          VARCHAR(20) NOT NULL,
kul_soyad       VARCHAR(20) NOT NULL,
kul_telefon     VARCHAR(11) NULL,
kul_email       VARCHAR(320) NOT NULL
);

```

---

Tablo oluştururken `kul_telefon` sütununda `NULL` değere izin verdik. Bunun sebebi; kayıt olacak kullanıcıların telefon numarasını vermek zorunda bırakmamaktır. Diğer bilgiler gereklidir, fakat telefon için bu gereklilik söz konusu değildir. Tabi ki isterseniz `kul_telefon` sütununu da `NOT NULL` tanımlayarak telefon bilgisini zorunlu hale getirebilirsiniz.

**RFC 2821**'e göre geçerli bir e-mail adresinin uzunluğu en fazla 320 karakterdir. Ve `VARCHAR` veri tipinin alabileceği maksimum alabileceği uzunluk 8000 bytes'dır. E-mail adresinin maksimum değerinin üzerinde bir değer atarsanız, bu gereksiz veri alanı kullanımı, yazılım uygulaması ile veri boyutu uyuşmazlığı ve injection saldırılarda güvenlik zafiyeti oluşturabilir. İyi bir geliştirici, ister son kullanıcı olsun, ister T-SQL sorgularını kullanacak geliştirici olsun, kullanıcının hata yapma ihtimalini hesap ederek geliştirme yapmalıdır. **Injection ile ilgili detaylı anlatım, sonraki veri güvenliğiyle ilgili bölümlerde anlatılacaktır.**

`diji_Kullanicilar` tablosunu hakkında yapısal bilgileri öğrenelim.

```
EXECUTE sp_help 'diji_Kullanicilar';
```

---

## ALTER İLE NESNELERİ DEĞİŞTİRMEK

`ALTER` ifadesi, `CREATE` ifadesiyle oluşturulan nesneler üzerinde yapısal değişiklikler yapılabilmesini sağlar.

```
ALTER <nesne_tipi> <nesne_adi>
```

---

## VERİTABANINI DEĞİŞTİRMEK

`ALTER` ifadesiyle, `DIJIBIL` veritabanınızın yapısında bazı değişiklikler yaparak inceleyelim.

Veritabanı üzerinde değişiklik yapmadan önce yapısını inceleyelim;

```
EXEC sp_helpdb DIJIBIL;
```

---

Biz **DIJIBIL** veritabanını bir test ve uygulama veritabanı olarak hazırladık. Bu nedenle boyutlarını çok küçük tuttuk. Şimdi bu veritabanına dışarıdan 250MB'lık bir veri ekleyeceğimizi düşünelim. Veritabanımızın veri dosyası 21MB büyüklüğünde ve **FILEGROWTH**, yani tek seferde boyut büyütme genişletme değerimiz 16MB idi. 250MB'lık veri eklemesini yaparken, 21MB'lık veritabanımız 16MB'lik parçalar halinde, defalarca genişletme işlemine tabi olacaktır. Tek seferde yapılabilecek genişletme işlemi yerine, defalarca bu işlem yapılacaktır. Bu da performans açısından olumsuz bir durum teşkil eder. Doğru olan bu işlemin tek seferde yapılmasıdır.

Şimdi, veritabanı boyutunu genişletmek için **DIJIBIL** veritabanımızın yapısında değişiklik yapalım.

**DIJIBIL** veritabanımızın veri depolama kapasitesini artırmak için veri dosyasının (MDF) genişliğini artırmamız gereklidir. Veritabanını oluştururken bu dosyaya verdığımız **NAME** bilgisine ihtiyacımız olacak. Biz veri dosyasının **NAME** bilgisine **DIJIBIL\_Data** adını vermiştık.

---

```
ALTER DATABASE DIJIBIL
MODIFY FILE
(
    NAME = DIJIBIL_Data,
    SIZE = 300MB
);
```

---

**ALTER** ifadesinde kullandığımız **NAME** bilgisi veri dosyamızın adını içerir. Bu soru ile **DIJIBIL** veritabanına ait **DIJIBIL\_Data** dosyasının (veri dosyamızın adı), **SIZE** yani dosya büyülüğu değerini 300MB yapmak istediğimizi bildirerek gerekli genişletme işlemini yapmış oluyoruz.

Bu sorgunun sonucunda fiziksel dosyaların yeni bilgilerini ilgili klasörden inceleyelim.

 DIJIBIL_Data.mdf	307.200 KB
 DIJIBIL_Log.ldf	2.048 KB

Şimdi de, yeni veritabanı boyutumuzu `sp_helpdb` sistem prosedürümüzle görelim.

---

```
EXEC sp_helpdb DIJIBIL;
```

---

	name	db_size	owner	dbid	created	status	compatibility_level	
1	DIJIBIL	302.00 MB	Cihanozhan\Cihan Ozhan	9	Oct 19 2012	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	110	
	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	DIJIBIL_data	1	C:\Databases\DIJIBIL_Data.mdf	PRIMARY	307200 KB	Unlimited	16384 KB	data only
2	DIJIBIL_Log	2	C:\Databases\DIJIBIL_Log.ldf	NULL	2048 KB	2147483648 KB	16384 KB	log only

Veritabanı genişletme ile ilgili önemli bir husus da şudur; **DIJIBIL** veritabanımızın **MAXSIZE** parametre değeri **UNLIMITED** olarak bildirildi. Ancak bu değer 300MB'den düşük olsaydı da bu sorgu çalışlığında hata vermeyecekti. Bunun sebebi veritabanı boyutunu bilinçli bir şekilde, açıkça değiştirmiş olmamızdır. Bu durumda veritabanının **MAXSIZE** değerini de yeni boyutlandırmaya göre yeniden güncelleyp, yükseltmeniz gerekecekti.

Eğer **MAXSIZE** olarak **UNLIMITED** değil de, 100MB vermiş olsaydık ve veritabanının **MAXSIZE** değerine kadar **FILEGROWTH** değeri ile otomatik olarak SQL Server'ın kendisinin artırmasını bekleyerdik, 250MB veri eklemeye çalıştığımızda **MAXSIZE** değerini aşamayacak ve verilerin kalan kısmı bu sınırlamadan dolayı eklenmeyecekti.

Ayrıca gene **MAXSIZE** ile 100MB sınırlaması getirdiğimiz bir veritabanına yukarıda yaptığımız gibi T-SQL kullanarak veritabanı genişletme işlemi uyguladığımızda yeni boyut (300MB), **MAXSIZE**'nın üzerinde olacağı için yeni **MAXSIZE** değeri en son yaptığımız genişletme işlemindeki değer olacaktır. Ve veritabanı dolmuş olacağı için yeniden bir **MAXSIZE** tanımlaması yapmanız gerekecektir.

## VERİTABANI TABLOSUNU DEĞİŞTİRMEK

SQL Server'ın temel nesnelerinden olan tablolar en fazla işleme tabii tutulan nesnelerdir. Veri ile ilgili tüm işlemler ve nesnelerin tablolar üzerinden gerçekleştirilmesi, zaman içerisinde veri ya da çeşitli yapılandırma gereksinimleri nedeniyle, birçok düzenleme ve yapısal değişiklik yapılmasını zorunlu hale getirir.

Bu işlem için `diji_Kullanicilar` tablosunu kullanacağız. Öncelikle tablomuzun yapısını inceleyelim.

---

```
EXEC sp_help diji_Kullanicilar;
```

---

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	kul_ID	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	kul_ad	varchar	no	20			no	no	no	Turkish_CI_AS
3	kul_soyad	varchar	no	20			no	no	no	Turkish_CI_AS
4	kul_telefon	varchar	no	11			yes	no	yes	Turkish_CI_AS
5	kul_email	varchar	no	320			no	no	no	Turkish_CI_AS

Bu sorguyu çalıştırıldığınızda ekranınıza 5 ayrı sonuç listelenecektir. Ben size, tablomuz ile doğrudan ilgili olan 2. sonucu gösteriyorum.

Şimdi, tablomuzda kullanıcılarımızın adres ve kayıt tarih bilgilerini saklamak için `kul_adres` ve `kul_kayitTarih` sütunları ekleyelim.

---

```
ALTER TABLE diji_Kullanicilar
ADD
kul_adres      VARCHAR(150) NULL,
kul_kayitTarih DATETIME      NOT NULL DEFAULT GETDATE();
```

---

Bu sorgu ile birlikte artık tablomuza `VARCHAR` veri tipinde ve 150 karakterlik veri alabilen `kul_adres` sütunumuz ve kullanıcı kayıt bilgisini tutan `kul_kayitTarih` olmak üzere iki sütunumuz eklenmiş oldu. `kul_adres` sütunumuz `NULL` geçilebilir, fakat `kul_kayitTarih` sütunumuza veri girişi yapılması `NOT NULL` ile zorunlu tutulmaktadır. Eğer yazılım ya da son kullanıcı tarafından veritabanına kayıt tarih bilgisi gönderilmez ise `DEFAULT` ile belirttiğimiz `GETDATE()` sistem fonksiyonu, sistemin o anki zaman bilgisini ekleyerek boş bırakılmamasını garanti edecektir.

## DROP İLE NESNE SİLMEK

SQL Server'da nesneleri silmek için `DROP` kullanılır. `DROP`'un amacı veri silmek değil, nesnelerin kaldırılmasını (silinmesi) sağlamaktır.

---

```
DROP <nesne_tipi> <nesne_ismi>
```

---

## BİR VERİTABANI TABLOSUNU SİLMEK

Veritabanında bir tablo silmek için, tek satırlık `DROP` sorgusu yeterlidir. Bunun için;

---

```
DROP TABLE <tablo_ismi>
```

---

**DIJIBİL** veritabanımızda silme işlemi için oluşturduğum `diji_Kitaplar` tablosunu silelim.

---

```
USE DIJIBİL                               Command(s) completed successfully.  
GO  
DROP TABLE diji_Kitaplar;
```

---

Burada, silme işlemini gerçekleştiren komut aslında sadece 3. satırdaki `DROP` ile başlayan komuttur.

`USE DIJIBİL` yazılmasının sebebi; yukarıdaki **Available Databases** bölümünde, farklı ve bu silme işlemini yapacağımız veritabanıyla aynı tablolara sahip bir veritabanı seçiliyse `DROP` başarıyla çalışacak ve bu tabloyu silecektir. Ancak bu silme işlemiyle, bizim istediğimiz veritabanında değil, farklı ve belki de önemli verilerin bulunduğu bir veritabanındaki kayıtların bulunduğu tabloyu silmiş olabiliriz. Bu nedenle, veritabanında her ne için ve ne silerseniz silin, bu işlemi 'nokta atışı' tabir edilecek şekilde net olarak belirtmek için `USE` ifadesiyle birlikte veritabanını belirtmenizi mutlaka öneririm.

Bu sorguda bulunan `GO` komutu da zorunlu değildir. `GO` olmadan da sorgunuz başarılı ve eksiksiz şekilde çalışacaktır. Ancak gelişmiş sorgu yapılarında, uzun kod bloklarında işlemlerin çalışma sıra ve anlaşılmaması gibi sebeplerden dolayı faydalı olacak bir kullanım alışkanlığıdır.

## BİR VERİTABANINI SİLMEK

Bir veritabanını silmek, bir tablo silmek kadar kolaydır. Bu işlem için gerekli sözdizimi;

---

```
DROP DATABASE <veritabani_ismi>
```

---

**KodLab** ismiyle oluşturduğumuz veritabanını silelim.

---

```
USE master
GO
DROP DATABASE KodLab;
```

---

Bu soru çalışlığında **KodLab** isimli veritabanı silinmiş olacaktır.

Veritabanı silme işlemi şu sebeplerden dolayı başarısız olabilir. Bunlar;

- SSMS'de geçerli olan bir veritabanını silmeye çalışmadığınıza emin olun.
- Silmeye çalışığınız veritabanına açılmış bir bağlantı olmadığına emin olun. Bunun için SSMS'yi kontrol edebileceğiniz gibi, **sp\_who** sistem prosedürüyle de öğrenebilirsiniz.

## **VERİ İŞLEME DİLİ**

### **(DML = DATA MANUPULATION LANGUAGE)**

Veri İşleme Dili (**DML**), tablolar üzerinde ekleme, güncelleme, seçme ve silme işlemlerini yapan ifadelerden oluşur. **INSERT** ile veri ekleme, **UPDATE** ile güncelleme, **SELECT** ile seçme ve görüntüleme, **DELETE** ile de veriyi silme işlemlerini gerçekleştirebiliriz. Veritabanı programlamanın veri ile ilgili kısmı temel olarak bu dört ifadeden oluşmaktadır.

Bu bölümde veri işleme dilinin temellerini anlatacağız.

### **INSERT İLE VERİ EKLEMEK**

Bir tabloya kayıt eklemek için kullanılır.

---

```
INSERT INTO tablo_ismi(sutun1_isim[,sutun2_ismi,...])
VALUES(deger1[,deger2,deger3,...])
```

---

**diji\_Kullanicilar** tablomuza yeni bir kullanıcı ekleyelim.

---

```
INSERT INTO diji_Kullanicilar
VALUES(1,'Cihan','Özhan',
      '02124537488','cihan.ozhan@dijibil.com',
      'İstanbul',GETDATE());
```

---

Bu sorguda tablo adının yanında herhangi bir sütun tanımlamasını yapmamamız dikkatinizi çekmiş olmalı. Sütun tanımlaması yapmak zorunda değiliz. Ancak bu durum bazı kurallara tabi tutulmuştur. Eğer yukarıdaki örnek gibi, herhangi bir sütun tanımlamadan veri girişi yapmak istiyorsanız; **VALUES** kısmında tablodaki tüm sütunlara, soldan sağa doğru sıralı şekilde veri eklemesi yapmamız gereklidir. Yani, sütun ismi belirtmeden veri giriyorsak SQL Server gerekli sütunların isimlerini ve değerlerini bizden isteyerek sorumluluğu bize bırakıyor.

Şimdi de, sütun isimleriyle birlikte yeni bir kayıt ekleyelim.

---

```
USE DIJIBIL
GO
INSERT INTO diji_Kullanicilar(
kul_ID, kul_ad, kul_soyad,
kul_telefon,kul_email,kul_adres,
kul_kayitTarih
)
VALUES(2, 'Kerim','Fırat','02124532122',
'kerim.firat@dijibil.com','İstanbul',GETDATE());
```

---

Bu şekilde de başarılı bir kayıt ekleme işlemi gerçekleşecektir. Sütun isimleriyle **INSERT** işleminde dikkat edilmesi gereken, sütun isimleriyle bu sütunlara vereceğimiz değerlerin, aynı sırayla veritabanına gönderilmesi gerektidir.

İki soru tipi için de ortak bir özellik de, tablomuzda bulunan **kul\_ID** sütunlarını sorgumuzda yazmamış olmamızdır. Bunun nedeni, **kul\_ID** sütunu **PRIMARY** ve **IDENTITY** sütun olmasıdır. Bu özelliklere sahip sütun, kendi değerini otomatik olarak ürettiği için bizim herhangi bir değer vermemize gerek yoktur. Eğer **kul\_ID** sütununa değer gönderseydik, sorgu çalışmaz ve hata üretecekti.

**INSERT** sorgunuzun sütun kısmına **kul\_ID**, değer kısmına da ilk değer olarak bir sayısal veri ekleyerek sorguyu çalıştırıp, çıkan hata sonucunu inceleyebilirsiniz.

## SELECT İLE VERİ SEÇMEK

Bir tabloda bulunan kayıtları seçmek için kullanılır. Veritabanı programlamanın veri işlemlerinde en çok kullanılan ve en karmaşık hallerde sorgular geliştirilen özellikleidir. Konular ilerledikçe neden **SELECT** ifadesinin daha çok kullanıldığı fark edeceksiniz.

---

```
SELECT sutun_ismi1[,sutun_ismi2,...]
FROM tablo_ismi
```

---

En temel **SELECT** sorgumuz ile önceki **INSERT** konusunda girdiğimiz kayıtları seçelim.

---

```
USE DIJIBIL
GO
SELECT * FROM diji_Kullanicilar;
```

---

	kul_ID	kul_ad	kul_soyad	kul_telefon	kul_email	kul_adres	kul_kayitTarih
1	1	Kerim	Frat	02124531234	kerim.firat@dijibil.com	Istanbul	2013-01-10 00:16:01.187
2	2	Cihan	Ozhan	02123456789	cihan.ozhan@dijibil.com	Istanbul	2013-01-10 00:16:04.637

Sorguda kullandığımız yıldız (\*) işaretinin anlamı, belirtilen tabloda bulunan tüm sütunların seçilmek istenmesidir. Yıldız işaretini bizi tüm sütunları tek tek yazmaktan kurtardığı için faydalıdır diyebiliriz. Fakat, yıldız işaretinin önemi dezavantajları vardır. Dezavantajlarını anlayabilmek için öncelikle nasıl çalıştığını inceleyelim.

Yıldız işaretile hazırladığımız sorgu veritabanı motoruna gönderilir. Veritabanı motoru yıldız (\*) işaretinin ne olduğunu, hangi sütunları ifade ettiğini anlamak için sorgu sonunda bulunan tablo adının yapısını sorgulayarak sistemde bu tabloya (*diji\_Kullanicilar*) ait tüm sütunların isimlerini elde eder. Daha sonra bu isimleri açık **SELECT** sorgusu halinde sıraya sokar ve bu şekilde sorguyu çalıştırarak sonucu ekrana getirir. Bu işlem, veritabanına ek sorgular gerçekleştirmesini zorunlu hale getirdiği için ağ trafiği, veritabanı gücünün gereksiz kullanımı, sorgular çoğaldığı için de sorgunun daha uzun süremesi gibi bir çok performans kaybını beraberinde getirir. Ayrıca bir tabloda genel olarak tüm sütunları istemeyiz. Fakat yıldız (\*) kullanımında bize tüm sütunlar ve bu sütunlara ait tüm kayıtlar getirileceği için yıldız işaretinin dezavantajlarının yanında bir de gereksiz kayıtların getireceği performans kaybı da eklenecektir. Bu nedenle yıldız (\*) işaretini profesyonel uygulamalarda kullanmamanız kesinlikle tavsiye edilir. Bu işaretin kullanılabilirliğiniz alanlar test ve geliştirme süreçleri olmalıdır.

Yıldız (\*) işaretiyile hazırladığımız sorguyu açık sütun isimleriyle tekrar hazırlayalım.

---

```
USE DIJIBIL
GO
SELECT
    kul_ID,kul_ad,kul_soyad,kul_telefon,
    kul_email,kul_adres,kul_kayitTarih
FROM
    diji_Kullanicilar;
```

---

Bu sorgu sonucunda gelen kayıt ile yıldız (\*) işaretiyile gelen kayıtlar tamamen aynıdır. Ancak performans olarak bu sorgulama yöntemi yıldız (\*) işaretliye göre daha performanslıdır.

## UPDATE İLE VERİ GÜNCELLEMEK

Tabloda satır ya da satırların değerlerini değiştirir-günceller.

---

```
UPDATE tablo_ismi
SET alan = deger
WHERE sart_tanimlari
```

---

Tablomuzda **ID** değeri 3 olan kullanıcının adres bilgisini “*İstanbul / Fatih*” olarak değiştirelim.

---

```
UPDATE diji_Kullanicilar
SET kul_adres = 'İstanbul / Fatih'
WHERE kul_ID = 2;
```

---

2	2	Kerim	Fırat	02124532122	kerim.fırat@dijibil.com	İstanbul / Fatih	2013-02-18 19:30:32.267
---	---	-------	-------	-------------	-------------------------	------------------	-------------------------

**UPDATE** işleminde güncellenecek kayıt konusunda dikkatli olmaliyiz. Standart olarak **UPDATE** ifadesi, **WHERE** filtreleme ifadesi olmadan da çalışacaktır. Ancak **WHERE** olmadan çalışan bu sorgu, tablodaki tüm kayıtların adres bilgilerini değiştirecektir.

## **DELETE İLE VERİ SİLMEK**

Tabloda bulunan verileri silmek için kullanılır. **UPDATE** işleminde olduğu gibi, **DELETE** sorgusunda da hangi verilerin silinmek istediği **WHERE** ifadesiyle belirtilmelidir. **WHERE** ifadesi kullanılmayan **DELETE** sorgusu, tablodaki tüm kayıtların silinmesine neden olacaktır.

---

```
DELETE FROM tablo_adi
WHERE sart_tanimlari
```

---

diji\_Kullanicilar tablomuzdaki 3 id'li kaydı silelim.

---

```
DELETE FROM diji_Kullanicilar WHERE kul_ID = 2;
```

---

Sizin tablonuzda kayıt ekleme ve silme denemelerinize göre 3 id'li kayıt olmayabilir. Farklı bir id bilgisine sahip kaydı silebilirsiniz.

**DELETE** sorgumuzda **WHERE** filtresi kullanmadığımızda oluşacak sonuçları kendiniz test etmek için;

---

```
DELETE FROM diji_Kullanicilar;
```

---

Bu sorguyu çalıştırıldığınızda ilgili tablodaki tüm kayıtların silindiğini göreceksiniz. Bu nedenle, **DELETE** sorgusunu kullanırken sonuçlarını tekrar gözden geçirmelisiniz.

Bu ifade de dikkat edilmesi gereken bir nokta da sileceğiniz verinin türüdür. Benzersiz (*unique*) bir değer silecekseniz, bu silme işlemi sonucunda tek kayıt silineceği kesindir. Ancak adı C harfi ile başlayan kullanıcıları silmek istediğinizde bu işlem sonucunda kaç kayıt silineğini tahmin edemeyebilirsiniz. Bu nedenle, **DELETE** ve **UPDATE** ifadesini kullanırken kapsam hesaplaması yapılmalıdır.

## **VERİ KONTROL DİLİ**

### **(DCL = DATA CONTROL LANGUAGE)**

Veri Kontrol Dili (DCL), bir veritabanına erişecek kullanıcıları ve rollerin izinlerini değiştirmek için kullanılır.

- GRANT** Kullanıcıların verileri kullanmasına ve T-SQL komutlarını çalıştırmasına izin verir.
- DENY** Kullanıcıların verileri kullanmasını kısıtlar.
- REVOKE** Daha önce yapılan tüm kısıtlama ve izinleri iptal eder.

SQL Server'da DCL komutlarını kullanabilmek için varsayılan olarak yetki sahibi olan gruplar;

- **sysadmin**
- **dbcreator**
- **db\_owner**
- **db\_securityadmin**

Veritabanı sunucusuna dışarıdan erişebilmek için bir login oluşturulmalıdır. **DIJILogin** isminde bir login oluşturalım.

---

```
CREATE LOGIN DIJILogin WITH PASSWORD = 'login_sifre';
```

---

Oluşturduğumuz **DIJILogin** isimli login ile sunucuya erişebilmemiz için bir User (Kullanıcı) tanımlayalım.

---

```
CREATE USER DIJIUser FOR LOGIN DIJILogin;
```

---

Kullanıcı (*User*) ve giriş (*Login*) tanımlamalarımızı aynı isimler ile yaptıysak, kullanıcı tanımlarken kullandığımız **FOR LOGIN** ifadesine gerek yoktur.

---

```
CREATE APPLICATION ROLE AppRole
WITH PASSWORD = 'app_role_pwd',
DEFAULT_SCHEMA= AppRole;
```

---

## GRANT İLE YETKİ VERMEK

Veritabanı kullanıcısına, veritabanı rolü ya da uygulama rolüne izinler vermek için kullanılan komuttur.

---

```
GRANT <ALL veya izinler>
ON <izin_verilenler>
TO <hesaplar>
```

---

**ALL** ifadesi, tüm hakları vermek için kullanılır.

**DIJIUser** kullanıcısına tablo oluşturma izni verelim.

---

```
GRANT CREATE TABLE  
TO DIJIUser;
```

---

Aynı anda **AppRole** isimli uygulama rolümüzü de yetkilendirebiliriz.

---

```
GRANT CREATE TABLE TO AppRole, DIJIUser;
```

---

Yeni bir kullanıcı oluşturarak hem veritabanı, hem de tablo oluşturma izni verelim.

**newDijiUser** isimli yeni kullanıcımızı oluşturalım.

---

```
CREATE USER newDijiUser FOR LOGIN DIJILogin;
```

---

**newDijiUser** kullanıcımıza veritabanı ve tablo oluşturma izni verelim.

---

```
GRANT CREATE DATABASE, CREATE TABLE TO newDijiUser;
```

---

## WITH GRANT OPTION İLE BASAMAKLI YETKILENDİRME

Bu deyim ile izin verilen bir kullanıcının bu nesne üzerinde aldığı izni bir başka kullanıcıya verebilmesi için kullanılır.

---

```
GRANT SELECT, INSERT ON diji_Kullanicilar  
TO AppRole  
WITH GRANT OPTION;
```

---

Bu durumda **AppRole** ile belirtilen role sahip herkes, **diji\_Kullanicilar** tablosu üzerinde başkalarına da **SELECT** ve **INSERT** erişim izni tanımlama hakkına sahip olacaktır.

## DENY İLE ERIŞİMİ KISITLAMAK

Kullanıcıların erişimlerini kısıtlamak için kullanılır.

---

```
DENY { ALL veya izinler} TO {kullanıcılar}
```

---

**DIJIUser** isimli kullanıcımızın tablo oluşturma yetkisini kaldıralım.

---

```
DENY CREATE TABLE TO DIJIUser;
```

---

**DIJIUser** isimli kullanıcımızın **diji\_Kullanicilar** tablosunda **SELECT** işlemi yapmasını engelleyelim.

---

```
DENY SELECT ON diji_Kullanicilar TO DIJIUser;
```

---

## REVOKE İLE ERIŞİM TANIMINI KALDIRMAK

Tüm kısıtlama ve izinleri iptal etmek için kullanılır. Bir nesneyi oluşturan kullanıcının, nesne üzerindeki yetkilendirme ve kullanım hakkı iptal edilemez.

**REVOKE** komutunu, **sys\_admin** rolü ya da **db\_owner**, **db\_securityadmin** sabit veritabanı rollerine sahip kullanıcılar ve nesne için, **dbo** olan kullanıcı çalıştırabilir.

---

```
REVOKE {ALL veya izinler} {TO veya FROM} {hesaplar}
```

---

**PUBLIC** rolüne verilen tüm yetkileri kaldıralım.

---

```
REVOKE ALL TO PUBLIC
```

---

**DIJIUser** kullanıcımıza uyguladığımız **diji\_Kullanicilar** tablosunda **SELECT** işlemini yapamama kısıtlamasını kaldıralım.

---

```
REVOKE SELECT ON diji_Kullanicilar TO DIJIUser;
```

---

# VERİLERİ SORGULAMAK

3

Veritabanının başlıca amacı; ilişkisel olarak sakladığı veriyi doğru, hızlı, güvenli ve en iyi performans ile sorgulayarak doğru sonuçları vermektedir. Tüm veritabanı yönetim sistemlerinde en çok kullanılan sorgular veriyi seçme ve gösterme ile ilgili olan sorgu yapılarıdır.

**INSERT**, **UPDATE** ya da **DELETE** gibi sorguların yapacakları işler kısıtlıdır. Çünkü işlevsel olarak az sayıda görevde sahiptirler. Ancak **SELECT** sorgusu bir veritabanı uygulamasında sayısız defa kullanılan bir sorgu yapısının temelidir. Bir veriyi gösterirken, formatlarken, sütunları seçerken, filtreleme ve koşullar belirleme, geçici ya da uzaysal veriler ile çalışmak gibi daha birçok iş için kullanılan sorgudur.

Verileri sorgulamak adındaki bu bölümde ilişkisel verileri sorgulayarak, ihtiyaç dahilinde doğru veriyi doğru yöntem ile elde etmeyi öğreneceksiniz.

## OPERATÖR TÜRLERİ

T-SQL ile programatik işlemler yapabilmek için operatörler kullanılır. Bu operatörlerin bazıları şu şekilde isimlendirilir.

- Karşılaştırma Operatörleri
- Mantıksal Operatörler
- Aritmetik Operatörleri
- Atama Operatörü
- Metin Birleştirme Operatörü

## ARİTMETİK OPERATÖRLERİ

SQL Server'da T-SQL ile matematiksel hesaplama işlemlerini gerçekleştiren operatörlerdir.

Aritmetik operatörleri şunlardır;

Operatör	Anlamı
%	Mod Alma
*	Çarpma
/	Bölme
+	Toplama
-	Çıkarma

Aritmetik operatörler tüm programlama dilleri ve veritabanı sorgu dillerinde mevcuttur. Hepsinde bir öncelik sıralaması vardır. Bu sıralama işlemi yukarıdaki sıralamaya göre yukarıdan aşağıya doğru tüm dillerde aynıdır.

5 ile 3 sayılarını toplayarak ekranda görüntüleyelim.

```
SELECT 5 + 3;
```

Hesaplama işlemlerinde parantez kullanımına örnek olarak daha karmaşık bir hesap yapalım.

---

```
SELECT ((5 + 3) * 6) * 2; -- Bu işlem sonucu 96 olacaktır.
```

---

Hesaplama işlemlerini birçok sorgu ifadesiyle birlikte kullanabilirsiniz. Sadece veriyi seçme işlemlerinde değil, stored procedure ya da benzeri özel işlemler yapacağınız nesneler içerisinde de matematiksel işlemlerde kullanabilirsiniz.

## ATAMA OPERATÖRÜ

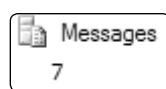
Eşit (=) işaretıyla ifade edilir. Değişkenlere değer atarken kullanılır.

Degisken adında bir değişken tanımlayıp SET ve atama operatörü (=) kullanarak bir değer atayalım ve ekranda görüntüleyelim.

---

```
DECLARE @Degisken VARCHAR(10);
SET @Degisken = 7;
PRINT @Degisken;
```

---



## METİN BİRLEŞTİRME OPERATÖRÜ

Artı (+) işaretiyile ifade edilir. Birden fazla **string** değeri yan yana birleştirerek, yazmak için kullanılan operatördür. Yan yana getirilen bu kayıtlar gerçekten birleştirilmez, birleşik olarak görüntülemeyi sağlar.

---

```
SELECT 'DIJIBIL.com' + ' & ' + 'KodLab.com';
```

---

(No column name)
1 DIJIBIL.com & KodLab.com

Yukarıdaki sorgu sonucunda, tek tırnaklar arasındaki **string** değerler birleştirilerek ekranda görüntülenecektir.

## SELECT İLE KAYITLARI SEÇMEK

Kayıtları seçmek (*Selecting*) ve listelemek için kullanılan **SELECT** sorgu yapısını, **T-SQL'e Genel Bakış** bölümünde incelemiştik. Şimdi bir hatırlatma yaparak **SELECT** sorgularının nasıl daha etkili kullanılacağına göz atalım.

**SELECT** sorgularında tüm tabloyu seçmek için \* (*yıldız*) işaretini kullanılır. Bu işaret performans açısından verimli olmadığı gibi tüm sütunları getirdiği için ihtiyaç fazlası işlem yapıyor da diyebiliriz.

---

```
SELECT * FROM Production.Product;
```

---

	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

Bu işlemi kısıtlayarak daha az sütunu kendi belirttiğimiz şekilde listelemek istersek;

---

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

---

	ProductID	Name
1	1004	% 20 indirimli ürün
2	1	Adjustable Race
3	461	Advanced SQL Server
4	879	All-Purpose Bike Stand
5	712	AWC Logo Cap
6	3	BB Ball Bearing
7	2	Bearing Ball

**SELECT** sorgularında sütunların listeleme sırasını belirlemek bizim elimizdedir. Tabloda ilk sırada yer alan **ProductID** sütununu dilersek **SELECT** sorgumuzda son sırada listeleyebilir ya da sorgumuza dahil etmeyebiliriz.

Yaygın bir kullanımı olmasa da şu şekilde de **SELECT** sorgusu oluşturulabilir.

---

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

---

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

Bu sorgu ile, önce tablodaki **Name** sütunu, sonra \* kullanımı ile **Name** dahil tüm sütunlar listelenir.

## DISTINCT İLE TEKİLE İNDİRGENEK

Aynı veriye sahip sütunları sorgularken, sonuç kümesinde her kayıttan sadece bir tane olmasını, yani tekrarlamayı engellemek istiyorsak **DISTINCT** kullanılır.

Personellerin isimlerini, her isimden sadece bir tane olacak şekilde listeleyelim.

---

```
SELECT DISTINCT FirstName FROM Person.Person;
```

---

	FirstName
1	Gustavo
2	Sheena
3	Henry
4	Claudia
5	Yvonne
6	Priscilla
7	Scot

## UNION VE UNION ALL İLE SORGU SONUÇLARINI BİRLEŞTİRMEK

Bazen **SELECT** sorgularında, birden fazla sorgunun sonucunu birleştirmek ve birleştirilen sorgular üzerinde farklı sorgular oluşturmak gerekebilir.

Bu tür birleştirme işlemleri için **UNION** ve **UNION ALL** kullanılır.

---

```
SELECT BusinessEntityID, ModifiedDate, SalesQuota
    FROM Sales.SalesPerson
   WHERE SalesQuota > 0
UNION
SELECT BusinessEntityID, QuotaDate, SalesQuota
    FROM Sales.SalesPersonQuotaHistory
   WHERE SalesQuota > 0
ORDER BY BusinessEntityID DESC, ModifiedDate DESC;
```

---

Yukarıdaki sorguda iki ayrı sorgu bulunmaktadır. Bu sorgular farklı tablolardan farklı verilerin listelenmesi için kullanılmıştır. Ancak bu sorguların sonucunda listlenecek kayıtların birleştirilerek listelenmesini istiyoruz.

İlk sorguda **Sales.SalesPerson** tablosunda, **SalesQuota** sütun değeri 0'dan büyük olanları, ikinci sorguda ise **Sales.SalesPersonQuotaHistory** tablosundaki bazı koşullara uyan satırların listelenmesini istedik. Son olarak ise, bu kayıtların tamamının birleştirildikten sonra listelenmesi için iki sorgu arasında **UNION** kullandık.

**UNION** operatörü, ilk sorguda var olan ve ikinci sorguda tekrarlanacak kayıtları filtreler. Yani, sorgular için bir **DISTINCT** uygular. Bu şekilde, kayıt tekrarı olmaması garantilenmiş olur.

**UNION** ile **UNION ALL** operatörleri aynı işlem için kullanılır. **UNION ALL** operatörünün tek farkı, kayıtlar için bir **DISTINCT** uygulamamasıdır. Yani, **UNION** yerine **UNION ALL** kullanırsanız, ikinci sorguda yer alan kayıtlar herhangi bir kontrole tabi tutulmadan, birinci sorgu sonucunda aynı kayıtlar yer alsa bile ilk birleştirme işlemi gerçekleşir.

## WHERE İLE SORGU SONUÇLARINI FILTRELEMEK

Temel **SELECT** işlemlerini gerçekleştirerek birçok kayıt listeledik. Ancak, dikkat ederseniz bu kayıtlar belli koşullara göre hazırlanmış değildi. Tablodaki tüm sütunları ya da belirlediğimiz sütunlardaki tüm kayıtları listeleyecek sorgular kullandık.

Gerçek uygulamalarda bu tür sorgular yeterli değildir. Belirli koşulları sağlayan sonuçları isteyeceğimiz yüzlerce farklı istek ve sorunla karşılaşırız. Bu sorunları çözmek için kullanacağımız ve aslında **SELECT** sorgu gücünü tam anlamıyla kullanmamızı sağlayacak özelliği, verileri çeşitli koşullara göre filtreleyerek istediğimiz kayıtları listeleyebilmesidir.

### Söz Dizimi:

---

```
SELECT sutun_ismi1[, sutun_ismi2, ... | *]
FROM   tablo_ismi
WHERE  şart_ifadeleri
```

---

Ürünler tablosundan 5 no'lu ürünü getirmek, ürün isminin baş harfi A olan ürünleri getirmek gibi birçok farklı şekilde kullanılabilir.

## MANTIKSAL OPERATÖRLER

**AND**, **OR** ve **NOT** mantıksal operatörleriyle birden çok koşullu ya da birleşik listelemeleri gerçekleştirmek mümkün olmaktadır.

### AND

**SELECT** cümlelerinde **WHERE** filtresi ile birlikte kullanılır. İstenen kayıtların birden fazla özellikle belirtip filtrelenmesi işleminin gerçekleştirilemesini sağlar.

**Production.Product** tablosunda **ProductID** sütunu değeri 2 ve **Name** sütunu değeri "Bearing Ball" olan kayıtları listelemek için aşağıdaki gibi bir sorgu kullanılabilir.

---

```
SELECT ProductID, Name FROM Production.Product
WHERE ProductID = 2 AND Name = 'Bearing Ball';
```

---

	ProductID	Name
1	2	Bearing Ball

Bu sorgu ile istenen kayıtların **ProductID**'si 2 olması ve **Name** sütun değerinin de "Bearing Ball" olmasını garanti altına almış oluruz. Bu işlem tek satır üzerinde gerçekleştirilir. Yani **Production.Product** içerisindeki her kayıt için bu sorgulama yapılır ve her kayıttaki **ProductID** ile **Name** sütunları aynı satırda bu değerlere sahip olan kayıtlar sonuç olarak döndürülür.

**AND** operatörü ile 2 ya da daha fazla karşılaştırma da yapılabilir. Yukarıdaki sorgunun **WHERE** kısmını şu şekilde değiştirerek inceleyiniz.

---

```
WHERE ProductID = 2 AND Name = 'Bearing Ball' AND ProductNumber = 'BA-8327'
```

---

ProductID	Name
1	2

Sorgu sonucu aynı olacaktır. Ancak farklı kayıtlar ve işlemler için kullanıldığında daha anlamlı ve farklı sonuçlar üretilebilir.

## OR

**SELECT** cümlelerinde **WHERE** filtresi ile birlikte kullanılır. Birden fazla seçenek belirtip bu seçeneklerin herhangi birine uyan kayıtların getirilmesini sağlar. Satır bazlı değil, tablo bazlı çalışır.

**Production.Product** tablosunda **Name** sütunu değeri **Blade** ya da **Bearing Ball** olan kayıtları listelemek için aşağıdaki gibi bir sorgu kullanılabilir.

---

```
SELECT ProductID, Name FROM Production.Product
WHERE Name = 'Blade' OR Name = 'Bearing Ball';
```

---

ProductID	Name
1	2
2	Blade

Bu sorgu ile istenen kayıtların **Name** sütunu değeri **Blade** ya da **Bearing Ball** olması garanti altına alınmış olur. Tablo üzerinde **Name** sütunu **Blade** ve **Bearing Ball** olanlar getirilecektir.

**OR** operatörü ile 2 ya da daha fazla karşılaştırma da yapılabilir. Yukarıdaki sorgunun **WHERE** kısmını şu şekilde değiştirerek inceleyiniz.

---

```
WHERE Name = 'Blade' OR Name = 'Bearing Ball' OR Name = 'Chain'
```

---

	ProductID	Name
1	2	Bearing Ball
2	316	Blade
3	952	Chain

Sorgu sonucunda önceki gibi 2 değil, 3 kayıt dönecektir. Bir OR şartı gerçekleşmez ise diğer OR şartlarına bakılır ve sorgu ile eşleşen kayıtlar döndürülür.

Ürün listesinde seçme işlemi yapan bir kaydın OR 1=1 eklenderek devre dışı bırakılması.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    ProductID < 0 OR 1=1;
```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	2	Bearing Ball	BA-8327
3	3	BB Ball Bearing	BE-2349
4	4	Headset Ball Bearings	BE-2908
5	316	Blade	BL-2036
6	317	LL Crankarm	CA-5965
7	318	ML Crankarm	CA-6738

OR 1=1 komutu ,veritabanına yapılan hacking saldırıları için kullanılan en basit yöntemlerden biridir. Bu komut, eklenderek kendisinden önceki tüm şartlar sağlanmasa bile sorgunun kapsadığı tüm kayıtları getirecektir.



SELECT cümlelerinde WHERE filtresi ile birlikte kullanılır. Birlikte kullanıldığı operatörün olumsuzluk eki anlamına gelir. Bir koşul ile belirtilen kayıtların haricindeki kayıtları getirir. Örneğin; 10 sayılık bir veri kümesinde 1. ve 2. kayıtların koşul içerisinde NOT ile birlikte belirtildiği sorgu ifadelerinde bu kayıtlar hariç tüm kayıtlar (8 kayıt) getirilir.

## KARŞILAŞTIRMA OPERATÖRLERİ

İki değerin birbirileyile karşılaştırılması için kullanılır. Karşılaştırılan verilerin türü aynı olmalıdır. Bir **string**, bir başka **string** ile karşılaştırılabilir. Bir **string** ile sayısal bir tür karşılaştırılamaz.

Operatör	Anlamı
<	Küçük
>	Büyük
=	Eşit
< =	Küçük Eşit
> =	Büyük Eşit
! =	Eşit Değil
< >	Eşit Değil
LIKE	Metin Karşılaştırma Operatörü

**AdventureWorks** veritabanında bazı sorgular geliştirerek bu operatörleri inceleyelim.

**Production.Product** tablosunda **ProductID** değeri 5'den küçük olan kayıtları listeleyelim.

---

```
SELECT ProductID, Name FROM Production.Product WHERE ProductID < 5;
```

---

	ProductID	Name
1	1	Adjustable Race
2	2	Bearing Ball
3	3	BB Ball Bearing
4	4	Headset Ball Bearings

Aynı sorunu kullanarak, < işaretini > işaretini ile değiştirirseniz 5'den büyük kayıtları listeleyecektir.

**Production.Product** tablosunda **ProductID** değeri 4'e eşit olan kayıt ya da kayıtları listeleyelim.

---

```
SELECT ProductID, Name FROM Production.Product WHERE ProductID = 4;
```

---

	ProductID	Name
1	4	Headset Ball Bearings

`Production.Product` tablosunda `ProductID` değeri 316'dan küçük ve eşit olan kayıtları listeleyelim.

---

```
SELECT ProductID, Name FROM Production.Product WHERE ProductID <= 316;
```

---

	ProductID	Name
1	1	Adjustable Race
2	2	Bearing Ball
3	3	BB Ball Bearing
4	4	Headset Ball Bearings
5	316	Blade

Aynı soruları kullanarak `<=` işaretini `>=` işaretine değiştirdiğinizde, 316'ya eşit ve büyük olan kayıtları listeler.

`Production.Product` tablosunda `ProductID` değeri 316 olmayan kayıtları listeleyelim.

---

```
SELECT ProductID, Name FROM Production.Product WHERE ProductID <> 316;
```

---

	ProductID	Name
1	1004	% 20 indirimli ürün
2	1	Adjustable Race
3	461	Advanced SQL Server
4	879	All-Purpose Bike Stand
5	712	AWC Logo Cap
6	3	BB Ball Bearing
7	2	Bearing Ball

## LIKE

`LIKE` operatörü karakterler içerisinde karşılaştırma yapmak için kullanılır. `LIKE` (*Liken = Benzerlik Bulmak*) operatörü, karşılaştırma işlemini birçok farklı yöntem ile gerçekleştirebildiği için, basit metinsel arama işlemlerinde tercih edilebilecek yöntemdir.

`LIKE` ile etkili sorgular oluşturabilmek için joker karakterler kullanılmalıdır.

## JOKER KARAKTERLER

Sadece **LIKE** operatörü ile kullanılan joker karakterler, arama sorgularında bir ya da daha fazla harfin yerine geçer. Belli aralıklarda, belli harf ile başlayan ya da biten sorgularda joker karakterler kullanılır.

Joker karakterler	Anlamı
%	Birden fazla harf ve rakamın yerini tutar.
_	Bir tek harf ya da rakamın yerini tutar.
[Harf]	Herhangi bir harf yerine gelebilecek harfleri belirtir.
[^Harf]	Herhangi bir harf yerine gelemeyecek harfleri belirtir.
[A-Z]	A ile Z arasındaki harfleri belirtir.

`Production.Product` tablomuzda `Name` sütunu içerisinde “ad” karakterleri bulunan kayıtları listeleyelim.

---

```
SELECT ProductID, Name FROM Production.Product WHERE Name LIKE '%ad%'
```

---

	ProductID	Name
1	1	Adjustable Race
2	461	Advanced SQL Server
3	316	Blade
4	399	Head Tube
5	847	Headlights - Dual-Beam
6	848	Headlights - Weatherproof
7	4	Headset Ball Bearings

**LIKE** operatörü ile basit bir karşılaştırma işlemi yapalım.

Baş harfi “C” olan tüm kayıtların `ProductID`, `Name` ve `ProductNumber` sütunlarını getirelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE 'C%';
```

---

	ProductID	Name	ProductNumber
1	320	Chainring Bolts	CB-2903
2	321	Chainring Nut	CN-6137
3	322	Chainring	CR-7833
4	323	Crown Race	CR-9981
5	324	Chain Stays	CS-2812
6	504	Cup-Shaped Race	RA-2345
7	505	Cone-Shaped Race	RA-7490

**LIKE** sorgularında dönecek sonucu belirleyen etken, **LIKE**'dan sonra gelen tek tırnaklar içerisindeki joker karakterler ve bunların kombinasyonlarıdır.

Joker karakterleriyle ilgili birkaç farklı örnek yapalım.

“Be” ile başlayan tüm ürünlerin kayıtlarını listeleyelim.

---

```

SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE 'Be%';

```

---

	ProductID	Name	ProductNumber
1	2	Bearing Ball	BA-8327

Son 4 karakteri “karm” olan, yani “karm” ile biten kayıtları listeleyelim.

---

```

SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '%karm';

```

---

	ProductID	Name	ProductNumber
1	319	HL Crankarm	CA-7457
2	317	LL Crankarm	CA-5965
3	318	ML Crankarm	CA-6738

"hai" ifadesi içeren tüm kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '%hai%';
```

---

	ProductID	Name	ProductNumber
1	952	Chain	CH-0234
2	324	Chain Stays	CS-2812
3	322	Chainring	CR-7833
4	320	Chainring Bolts	CB-2903
5	321	Chainring Nut	CN-6137

'eflector' ile biten tüm 9 harfli kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '_eflector';
```

---

	ProductID	Name	ProductNumber
1	506	Reflector	RF-9198

İlk iki harfi belli olmayan, son harfleri 'flector' olan 9 harfli kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '__flector';
```

---

	ProductID	Name	ProductNumber
1	506	Reflector	RF-9198

'A'ya da 'c' ile başlayan tüm isimleri listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '[AC]%' ;
```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	320	Chainring Bolts	CB-2903
3	321	Chainring Nut	CN-6137
4	322	Chainring	CR-7833
5	323	Crown Race	CR-9981
6	324	Chain Stays	CS-2812
7	461	Advanced SQL Server	LR-2398

5 karakterli ve ilk karakteri 'A' ile 'c' arasında olan ve 'lade' ile biten tüm isimleri listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '[A-C]lade' ;
```

---

	ProductID	Name	ProductNumber
1	316	Blade	BL-2036

'A' ile başlayan ve ikinci karakteri c olmayan tüm isimler.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE 'A[^c]%' ;
```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	461	Advanced SQL Server	LR-2398
3	712	AWC Logo Cap	CA-1098
4	879	All-Purpose Bike Stand	ST-1401

Birden fazla karşılaştırma için **LIKE** kullanım örneği; 'a' ya da 'b' ya da 'c' ile başlayan ürünleri listeleyelim.

---

```

SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE 'A%' OR
    Name LIKE 'B%' OR
    Name LIKE 'C%';

```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	2	Bearing Ball	BA-8327
3	3	BB Ball Bearing	BE-2349
4	316	Blade	BL-2036
5	320	Chainring Bolts	CB-2903
6	321	Chainring Nut	CN-6137
7	322	Chainring	CR-7833

Karakter karşılaştırma işlemlerinde joker karakterlerin birçok faydasını gördük. Ancak bir metin içerisinde joker karakter bulunma olasılığı da yüksektir. Örneğin; yüzde (%) işaretti, birçok durumda kullanılan, bu işaretin yukarıda kullandığımız yöntemler ile sorgu içerisinde kullanamayız.

**LIKE** ile joker karakter karşılaştırması yapabilmek için farklı bazı yöntemler kullanılır.

İçerisinde yüzde (%) işaretini içeren kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '%[%]%' ;
```

---

	ProductID	Name	ProductNumber
1	1004	% 20 indirimli ürün	SK-9299

// karakteri bulunan kayıtları aramak istediğimizde ise, ek parametrelere gerek yoktur.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name LIKE '%/%' ;
```

---

	ProductID	Name	ProductNumber
1	523	LL Spindle/Axle	SD-2342
2	524	HL Spindle/Axle	SD-9872
3	873	Patch Kit/8 Patches	PK-7098
4	908	LL Mountain Seat/Saddle	SE-M236
5	909	ML Mountain Seat/Saddle	SE-M798
6	910	HL Mountain Seat/Saddle	SE-M940
7	911	LL Road Seat/Saddle	SE-R581

**LIKE** sorgularına olumsuzluk anlamı katmakta mümkündür. Bunun için **NOT** operatörü kullanılır.

İlk örneğimizde 'Be' ile başlayan kayıtları listelemiştik. Şimdi ise, **NOT** operatörünü kullanarak, 'Be' ile başlayan kayıtların olmadığı, yani 'Be' ile başlamayan tüm kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name, ProductNumber
FROM
    Production.Product
WHERE
    Name NOT LIKE 'Be%';
```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	3	BB Ball Bearing	BE-2349
3	4	Headset Ball Bearings	BE-2908
4	316	Blade	BL-2036
5	317	LL Crankarm	CA-5965
6	318	ML Crankarm	CA-6738
7	319	HL Crankarm	CA-7457

## BELİRLİ KAYITLAR ARASINDA SORGULAMA YAPMAK

**SELECT** ile **WHERE** koşulu oluştururken genel kullanım, belirli bir kaydı ya da bu kayıttan küçük, büyük ya da eşit olan kayıtların listelenmesidir. Ancak, bazen sadece belirli kayıtlar ya da belirli kayıtların arasındaki kayıtları listelemek isteyebiliriz. Bu gibi durumlarda kullanılacak aralık sorgulama tekniklerini inceleyelim.

### BETWEEN .. AND ..

**SELECT** cümlelerinde **WHERE** koşulu ile birlikte kullanılır. **Between** operatörü iki operand alır ve aldığı operandların aralığını kontrol eder. Between operatöründe ilk değer küçük, ikinci değer büyktür ve aldığı operandları da sorgulamaya dahil eder.

**Production.Product** tablosundaki **ProductID** sütun değeri 1 ile 4 arasında olan kayıtları getirelim.

---

```
SELECT ProductID, Name FROM Production.Product
WHERE ProductID BETWEEN 1 AND 4;
```

---

	ProductID	Name
1	1	Adjustable Race
2	2	Bearing Ball
3	3	BB Ball Bearing
4	4	Headset Ball Bearings

Sorgu sonucu dönen kayıtlardan görebileceğiniz gibi bu sorgu, 1 ile 4 değerlerini de kapsamaktadır.

`Production.Product` tablomuzdaki 320 ile 400 `ProductID` değerine sahip kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    ProductID BETWEEN 320 AND 400;
```

---

	ProductID	Name
1	320	Chainring Bolts
2	321	Chainring Nut
3	322	Chainring
4	323	Crown Race
5	324	Chain Stays
6	325	Decal 1
7	326	Decal 2

`Between` ile gerçekleştirdiğimiz işlemi, `AND` operatörü ile de yapabiliriz.

Bu örneğimizi farklı bir yöntem ile gerçekleştirelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    ProductID >= 320 AND ProductID <= 400;
```

---

	ProductID	Name
1	320	Chainring Bolts
2	321	Chainring Nut
3	322	Chainring
4	323	Crown Race
5	324	Chain Stays
6	325	Decal 1
7	326	Decal 2

**NOT** operatörünü kullanarak **BETWEEN** ile belirtilen kayıtları içermeyen sonuçları da listeleyebiliriz.

**BETWEEN** örneğimizin **WHERE** kısmını değiştirerek **NOT** işlemini uygulayalım.

---

```
ProductID NOT BETWEEN 320 AND 400;
```

---

### IN VE NOT IN

**IN** operatörü, sütun içerisinde bir ya da birden fazla belirli kayıt aramak için kullanılır. Aranmak istenen kayıtlar **IN** (bir, iki, ...) içerisinde virgül ile ayrılarak belirtilir.

**Production.Product** tablomuzda 1, 2, 3 ve 316 **ProductID** değerine sahip kayıtlarımızı listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    ProductID IN(1, 2, 3, 316);
```

---

	ProductID	Name
1	1	Adjustable Race
2	2	Bearing Ball
3	3	BB Ball Bearing
4	316	Blade

Sorgumuzun **IN()** kısmında belirtilen kayıtların veritabanında karşılığı olmadığı durumlarda, bulunamayan kayıtlar soru dışı bırakılır ve bulunan kayıtlar listelenir.

**IN** ile gerçekleştirdiğimiz soruyu, **OR** karşılaştırma operatörü ile de gerçekleştirebiliriz.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    ProductID = 1 OR
    ProductID = 2;
```

---

	ProductID	Name
1	1	Adjustable Race
2	2	Bearing Ball

**IN** operatörü ile metinsel arama da yapılabilir.

**Name** sütununda **chain** ve **Chairing** isimli ürünleri listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    Name IN('Chain', 'Chainring');
```

---

	ProductID	Name
1	952	Chain
2	322	Chainring

**IN** ile birlikte **NOT** operatörünü kullanarak belirtilen kayıtların dışındaki tüm kayıtlar listelenebilir.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    ProductID NOT IN(1, 2, 3, 316);
```

---

	ProductID	Name
1	1004	% 20 indirimli ürün
2	461	Advanced SQL Server
3	879	All-Purpose Bike Stand
4	712	AWC Logo Cap
5	877	Bike Wash - Dissolver
6	843	Cable Lock
7	952	Chain

`IN` operatörünü bir değeri birkaç farklı sütun içerisinde aramak için de kullanabiliriz.

'Chairing' değerini `Name` ve `Color` sütunları içerisinde arayarak eşleşen kayıtları listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    'Chainring' IN(Color, Name);
```

---

	ProductID	Name
1	322	Chainring

Aynı işlemi bir tarih aramak için de kullanabiliriz.

'2008-03-11 10:01:36.827' tarih değerini `SellStartDate` ve `ModifiedDate` sütunlarında arayarak sonuçları listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
WHERE
    '2008-03-11 10:01:36.827' IN(SellStartDate, ModifiedDate);
```

---

38.PNG

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

**AdventureWorks** veritabanında neredeyse tüm kayıtlar, aynı **ModifiedDate** değerine sahip olduğu için, bu sorgu sonucunda 500 civarında kayıt listelenecektir. Gerçek uygulamalarda saniye ve salise bilgisinin de aynı olduğu çok fazla kayıt bulunmaz.

## SQL SERVER'DA NULL VE BOŞLUK KAVRAMI

SQL Server'da boşluk ve **NULL** kavramı önemlidir. Bazen **NULL** değerleri görüntülemek isterken, bazen de bu değerleri manipüle ederek uygulamamızı hata ve **NULL** değer görünümünden kurtarmak isteriz. Bu ve bu gibi durumlarda müdahale edebilmek için boşluk ve **NULL** kavramına hakim olmalıyız.

### METINSEL DEĞERLER İLE NULL KULLANIMI

SQL Server'da metinsel bir değer ile **NULL**'ü birleştirmek varsayılan olarak mümkün değildir. İki metinsel değeri **NULL** ile birleştirerek bunu deneyelim.

---

```
SELECT 'Cihan' + NULL + 'Özhan';
```

---

	(No column name)
1	NULL

Sonuç olarak geri dönüşün **NULL** olduğunu görüyoruz. Peki gerçekten birleştirmemiz gerektiği bir durum söz konusu olursa ne yapmalıyız?

SQL Server her ne kadar bu tür durumlarda varsayılan destek sunmasa da bazı esnekliklere sahiptir.

Bir **NULL** ile metinsel değer birleştirme işlemi için aşağıdaki komutu çalıştırmanız yeterlidir.

---

```
SET CONCAT_NULL_YIELDS_NULL OFF;
```

---

Şimdi tekrar aynı birleştirme sorgusunu çalıştığımızda istediğimiz sonucun görüntülenmesini sağlamış oluruz.

---

```
SELECT 'Cihan' + NULL + 'Özhan';
```

---

(No column name)	
	1
	CihanÖzhan

Bu özelliğin varsayılan değerine geri dönmesi için `OFF` yerine `ON` yazmanız yeterlidir.

---

```
SET CONCAT_NULL_YIELDS_NULL ON;
```

---

## SPACE FONKSİYONU

SQL Server'da veriler arasında boşluk bırakmak için kullanılan fonksiyondur. Metinsel işlemlerde kullanmak için geliştirilmiştir. Nümerik bir değer ile metinsel bir değer birleştirilmesinde kullanılmaz. `SPACE` fonksiyonu, aldığı parametre değeri kadar boşluk karakteri oluşturur.

### Söz Dizimi:

```
SPACE( ifade );
```

İki metinsel değeri aralarında bir boşluk karakteri bırakarak birlestirelim.

---

```
SELECT 'Cihan' + SPACE(1) + 'Özhan';
```

---

(No column name)	
	1
	Cihan Özhan

10 karakter boşluk bırakmak için `SPACE(10)` yazmanız yeterlidir.

Aynı fonksiyonu nümerik değerler ile kullanırsak aralarına boşluk bırakarak birleştirmez, iki nümerik değeri toplayarak sonucu ekrana yazar.

---

```
SELECT 1 + SPACE(1) + 4;
```

---

(No column name)	
	1
	5

İki nümerik değer ile `SPACE` fonksiyonu kullanılırken değerlerden herhangi birinin tek tırnaklar içerisinde yazılmış olması bu değerlerin toplanmamasına

ve hata vermesine sebep olmaz. SQL Server varsayılan olarak tek tırnakları görmezden gelir ve nümerik bir değer olarak algılar.

Bir nümerik ile bir metinsel değer birleştirilmek istendiğinde ise hata verecektir.

---

```
SELECT 'DIJIBIL' + SPACE(5) + 2013;           -- Hata
```

---

Bu birleştirme işleminin gerçekleştirilebilmesi için tür dönüşümü gerektiğini bildiren bir hata alınır.

Conversion failed when converting the varchar value 'DIJIBIL' to data type int.

## SORGULARDA NULL DEĞER İŞLEMLERİ

Genelde SQL Server'a yeni başlayanlar için **NULL** kavramı zor ve yanlış anlaşılıbmaktadır. **NULL** değeri bir alanda boşluk dahil hiç bir verinin bulunmadığını belirtir. Çünkü boşluk bilgisayar dilinde bir değere sahiptir ve bir varlığa sahiptir. **NULL** kavramı bilinmezlik anlamına gelir ve **NULL** olan iki değer birbirile eşitlik ya da benzeri bir işleme tabi tutulamaz.

Geliştirilen yazılım ve raporlama gibi veritabanı kayıtlarının işlenerek yönetici uygulamalarda **NULL** kayıtlar, kullanılmadığı ya da doğru kullanılmadığı takdirde hem geliştirme aşamasında hatalara sebep olabilir, hem de son kullanıcı açısından **NULL** yazısının bir anlamı olmayacağı için **NULL** yerine farklı değerler gösterilerek kullanışlı bir işlev kazandırılabilir. Uygulamalarda görülen "Kayıt Bulunamadı" gibi metinler **NULL** kayıtların son kullanıcıya gösterilmesi için iyi bir örnek olabilir.

Microsoft, **NULL** ile ilgili işlemleri SQL Server'da yönetebilmek için bazı operatör ve fonksiyonlar geliştirmiştir.

## IS NULL OPERATÖRÜ

SQL Server'da sütunlar içerisinde **NULL** olan kayıtların birbirile eşitlik gibi sorgulama yapılamayacağını öğrenmiştim. Ancak sorgularımızda **NULL** kayıtları listeleyebilmek için geliştirilmiş bazı komutlar mevcuttur.

**IS NULL** operatörü **WHERE** filtresiyle birlikte kullanılarak **NULL** olan kayıtları listeler.

**Söz Dizimi:**


---

```
SELECT sutun_ismi1 [, sutun_ismi2, ...]
FROM   tablo_ismi
WHERE  sutun_ismi IS [NOT] NULL
```

---

**Production.Product** tablosundaki Size sütunu **NULL** olan kayıtları listeleyelim.

---

```
SELECT
  ProductID, Name, Size
FROM
  Production.Product
WHERE
  Size IS NULL;
```

---

	ProductID	Name	Size
1	1	Adjustable Race	NULL
2	2	Bearing Ball	NULL
3	3	BB Ball Bearing	NULL
4	4	Headset Ball Bearings	NULL
5	316	Blade	NULL
6	317	LL Crankam	NULL
7	318	ML Crankam	NULL

Sorgu sonucunda Size sütun değeri **NULL** olan kayıtlar listelenmiştir. Siz de bu örneği **Color** ve **Weight** sütunları üzerinde kullanarak test edebilirsiniz.

Aynı sorgunun **NOT** kullanımı için aşağıdaki şekilde değiştirmeniz yeterli olacaktır.

---

Size IS NOT NULL

---

	ProductID	Name	Size
1	680	HL Road Frame - Black, 58	58
2	706	HL Road Frame - Red, 58	58
3	709	Mountain Bike Socks, M	M
4	710	Mountain Bike Socks, L	L
5	713	Long-Sleeve Logo Jersey, S	S
6	714	Long-Sleeve Logo Jersey, M	M
7	715	Long-Sleeve Logo Jersey, L	L

## ISNULL FONKSİYONU

`NULL` değerler için kullanılan ve iki parametre alan `NULL` değer fonksiyonudur. İlk parametre olarak kontrol edilecek değeri alır ve `NULL` olduğu takdirde ikinci parametre olarak verilen değeri döndürür.

### Söz Dizimi:

---

```
ISNULL(kontrol_edilecek_deger, NULL_ise_verilecek_deger)
```

---

`Production.Product` tablomuzda `Color` sütunu üzerinde bir `NULL` kontrolü yapalım ve eğer kayıt `NULL` ise listelemeye "Renk Yok" yazarak listeletelim.

---

```
SELECT
    ProductID, Name, Color,
    ISNULL(Color, 'Renk Yok') AS Renk
FROM
    Production.Product;
```

---

	ProductID	Name	Color	Renk
1	1	Adjustable Race	NULL	Renk Yok
2	2	Bearing Ball	NULL	Renk Yok
3	3	BB Ball Bearing	NULL	Renk Yok
4	4	Headset Ball Bearings	NULL	Renk Yok
5	316	Blade	NULL	Renk Yok
6	317	LL Crankarm	Black	Black
7	318	ML Crankarm	Black	Black

`Production.Product` tablomuzda ürünlerin ağırlık bilgisini tutan `Weight` sütununu `ISNULL()` ile `NULL` kontrol işleme tabi tutalım. Eğer ağırlık bilgisi `NULL` ise sorgu sonucu olarak varsayılan değer 10 verelim.

---

```
SELECT
    Weight,
    ISNULL(Weight, 10)
FROM
    Production.Product;
```

---

	Weight	(No column name)
1	NULL	10.00
2	NULL	10.00
3	NULL	10.00
4	NULL	10.00
5	NULL	10.00
6	NULL	10.00
7	NULL	10.00

`ISNULL()` fonksiyonunu graplama fonksiyonları ile de kullanabiliriz. Örneğin; yukarıdaki sorgunun sonucunu bir `AVG()` fonksiyonuna aktaralım.

---

```
SELECT
    Weight,
    AVG(ISNULL(Weight, 10))
FROM
    Production.Product
GROUP BY Weight;
```

---

	Weight	(No column name)
1	NULL	10.000000
2	2.12	2.120000
3	2.16	2.160000
4	2.18	2.180000
5	2.20	2.200000
6	2.22	2.220000
7	2.24	2.240000

`AVG()` işleminde `Weight` değeri `NULL` olan kayıtların varsayılan olarak 10 olarak hesaplanması sağlanmış ve buna göre bir ortalama oluşturulmuştur.

`Sales.SpecialOffer` tablomuzda `Description`, `DiscountPct` ve `MinQty` sütunlarını getiren, `MaxQty` sütununu da `NULL` kontrolü yaparak, eğer `NULL` var ise, 0.00 varsayılan nümerik değeri gösterecek SQL sorgumuzu yazalım.

---

```

SELECT
    Description, DiscountPct, MinQty,
    ISNULL(MaxQty, 0.00) AS 'Max Miktar'
FROM
    Sales.SpecialOffer;

```

---

	Description	DiscountPct	MinQty	Max Miktar
1	No Discount	0,00	0	0
2	Volume Discount 11 to 14	0,02	11	14
3	Volume Discount 15 to 24	0,05	15	24
4	Volume Discount 25 to 40	0,10	25	40
5	Volume Discount 41 to 60	0,15	41	60
6	Volume Discount over 60	0,20	61	0
7	Mountain-100 Clearance Sale	0,35	0	0

## COALESCE FONKSİYONU

Aldığı parametrelerden **NULL** olmayan ilk ifadeyi döndürür. SQL Server' da yoğun olarak kullanılmayan fonksiyonlardandır.

### Söz Dizimi:

---

```
COALESCE( ifadel, ifade2 [ ...n ] )
```

---

**Production.Product** tablosunda bulunan **Color** sütunu değerleri üzerinde **COALESCE()** fonksiyonunu kullanalım. Eğer **Color** sütununda herhangi bir renk değeri girilmemiş ise "Renk Yok" yazmasını istiyoruz.

---

```

SELECT
    ProductID, Name,
    COALESCE(Color,'Renk Yok') AS Renk
FROM
    Production.Product;

```

---

ProductID	Name	Renk
1	Adjustable Race	Renk Yok
2	Bearing Ball	Renk Yok
3	BB Ball Bearing	Renk Yok
4	Headset Ball Bearings	Renk Yok
5	Blade	Renk Yok
6	LL Crankarm	Black
7	ML Crankarm	Black

Şimdi de `Person.Person` tablosundan `Title`, `FirstName`, `MiddleName` ve `LastName` sütunlarını getirelim. Ve bu sütunlardan `Title` ile `MiddleName` sütunlarına kayıt girilmemiş ise, boş sütunlarda Kayıt Yok yazsın.

---

```
SELECT
    COALESCE(Title,'Kayıt Yok'),
    FirstName,
    COALESCE(MiddleName,'Kayıt Yok'),
    LastName
FROM
    Person.Person;
```

---

	(No column name)	FirstName	(No column name)	LastName
1	Kayıt Yok	Ken	J	Sánchez
2	Kayıt Yok	Temi	Lee	Duffy
3	Kayıt Yok	Roberto	Kayıt Yok	Tamburello
4	Kayıt Yok	Rob	Kayıt Yok	Walters
5	Ms.	Gail	A	Erickson
6	Mr.	Jossef	H	Goldberg
7	Kayıt Yok	Dylan	A	Miller



`ISNULL()` fonksiyonundan bazı farkları vardır. `ISNULL()` fonksiyonunun aksine `COALESCE()` birden fazla parametre alabilir. Ayrıca `COALESCE()` fonksiyonu **ANSI** standartı olup, diğer veritabanı yönetim sistemlerinde de kullanılabilirken, `ISNULL()` SQL Server'a özgü bir fonksiyondur.

Şimdi `COALESCE()` fonksiyonunun birden fazla parametre ile nasıl çalıştığını bir örnek verelim.

`Production.Product` tablomuzdan `Name`, `Class`, `Color` ve `ProductNumber` sütunlarını çekiyoruz. Bu sütunların yanında `COALESCE()` fonksiyonunu kullanarak 3 parametreli bir istek yaptığımız 4. sütunu da ekliyoruz. Bu isteğimiz sonucunda 4. sütuna sahip oluyoruz. Bu sütuna ilk olarak `Class` sütunundan veri gelmesini eğer `Class` sütunu `NULL` ise `Color` sütunundan gelmesini, `Color` sütunu da `NULL` ise `ProductNumber` sütunundan veri getirilerek 4. sütunumuzun doldurulmasını istiyoruz.

---

```

SELECT
    Name, Class, Color, ProductNumber,
    COALESCE(Class,
        Color,
        ProductNumber) AS FirstNotNull
FROM Production.Product;

```

---

	Name	Class	Color	ProductNumber	FirstNotNull
1	Adjustable Race	NULL	NULL	AR-5381	AR-5381
2	Bearing Ball	NULL	NULL	BA-8327	BA-8327
3	BB Ball Bearing	NULL	NULL	BE-2349	BE-2349
4	Headset Ball Bearings	NULL	NULL	BE-2908	BE-2908
5	Blade	NULL	NULL	BL-2036	BL-2036
6	LL Crankarm	L	Black	CA-5965	L
7	ML Crankarm	M	Black	CA-6738	M

Bu işlemimizin sonucunda görüntülenecek kayıtların listeleme sırasında, karışmaması için aşağıdaki gibi fonksiyon içerisinde parametreleri **string** değerler ile birleştirerek daha anlaşılır sorgu sonucu elde edebiliriz.

---

```

SELECT
    Name, Class, Color, ProductNumber,
    COALESCE('Class : ' + Class,
        'Color : ' + Color,
        'ProductNumber : ' + ProductNumber) AS FirstNotNull
FROM Production.Product;

```

---

**NOT** NULL işlemlerinde genel olarak **COALESCE()** yerine **ISNULL()** kullanılmaktadır. Ancak işlem için birden fazla parametre alabilecek bir sorguya ihtiyacınız varsa, seçiminiz **COALESCE()** olmalıdır.

## NULLIF FONKSİYONU

İki parametre alan ve bu iki parametrenin değerleri aynı ise, geriye **NULL** değer döndüren fonksiyondur. Eğer parametrelerin değeri eşit değil ise ilk aldığı parametreyi geri döndürür.

### Söz Dizimi:

---

```
NULLIF( ifade1, ifade2 );
```

---

**Production.Product** tablomuzda **MakeFlag** ve **FinishedGoodsFlag** sütunlarını isteyip 3. sütun olarak da bu iki sütunu **NULLIF()** fonksiyonu ile karşılaştırıp eşit olup olmadığını kontrol ediyoruz. Eğer değerler eşit ise **NULL** değer, değil ise ilk parametrenin değerini döndürür.

---

```
SELECT
    MakeFlag, FinishedGoodsFlag,
    NULLIF(MakeFlag, FinishedGoodsFlag) AS 'Eşitse Null'
FROM
    Production.Product
WHERE
    ProductID < 10;
```

---

	MakeFlag	FinishedGoodsFlag	Eşitse Null
1	0	0	NULL
2	0	0	NULL
3	1	0	1
4	0	0	NULL

## SELECT İLE VERİLERİ SIRALAMAK

SQL Server'da veriler tablolar içerisinde sütunlarda yer alır. Tablolar ise belirli düzene sahip sütunların sıralı halde sorgulanmasıyla oluşur. Bir veri gösterim (*Select*) sorgusu gerçekleştirildiğinde herhangi bir özel istek belirtilmediği takdirde veriler, tablodaki sütunların sırasına göre listelenir. Bu, SQL Server'ın hızlı veri listelemek için kullandığı varsayılan bir yöntemdir. Ancak istediğimiz takdirde bu sıralamaya müdahale ederek, kendi belirlediğimiz sıraya göre listeleme işlemini gerçekleştirebiliriz. Bu işleme Sort(*Sorting* = Sınıflandırmak) denir.

## ORDER BY

SQL Server'da Sıralama işlemini gerçekleştiren komut **Order By**'dır.

### Söz Dizimi:

---

```
SELECT      sutun_isim1, sutun_isim2
FROM        table_name
ORDER BY    siralanacak_sutun [ASC|DESC];
```

---

**Production.Product** tablomuzdaki ürünleri Name sütununa göre listeleyelim.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
ORDER BY Name;
```

---

	ProductID	Name
1	1004	% 20 indirimli ürün
2	1	Adjustable Race
3	461	Advanced SQL Server
4	879	All-Purpose Bike Stand
5	712	AWC Logo Cap
6	3	BB Ball Bearing
7	2	Bearing Ball

Listelenen kayıtların sıralamasına dikkat edin. A'dan Z'ye doğru bir sıralama gerçekleştirildi. Bu, SQL Server'ın varsayılan sıralama algoritmasıdır.

SQL Server'da iki tip sıralama yöntemi vardır. Bunlar;

- **ASC (Ascending)**

SQL Server için varsayılan sıralama yöntemidir. Artan sırada listeleme yapar. Metinsel veri için A'dan Z'ye, nümerik veri için ise 0'dan 9'a doğru sıralama gerçekleştirir.

ASC tipinde sıralama yapmak için Order By'dan sonra özellikle bir belirtim yapmaya gerek yoktur. Ancak kod okunabilirliği açısından kullanılmasında fayda vardır.

• **DESC** (*Descending*)

ASC'nin tam tersidir. Azalan sırada listeleme yapar. Metinsel veri için Z'den A'ya, nümerik veri için ise 9'dan 0'a doğru sıralama gerçekleştirir.

`Production.Product` tablosundaki kayıtları `ProductID` sütununa göre artan şekilde sıralayalım.

```
SELECT
    ProductID, Name
FROM
    Production.Product
ORDER BY
    ProductID ASC;
```

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

`Production.Product` tablosundaki kayıtları `ProductID` sütununa göre azalan şekilde sıralayalım.

```
SELECT
    ProductID, Name
FROM
    Production.Product
ORDER BY
    ProductID DESC;
```

	ProductID	Name
1	1004	% 20 indirimli ürün
2	999	Road-750 Black, 52
3	998	Road-750 Black, 48
4	997	Road-750 Black, 44
5	996	HL Bottom Bracket
6	995	ML Bottom Bracket
7	994	LL Bottom Bracket

**Order By** ile birden fazla sütuna göre de sıralama gerçekleştirebiliriz.

---

```
SELECT
    ReorderPoint, Name,
    ProductID
FROM
    Production.Product
ORDER BY
    ReorderPoint, Name ASC;
```

---

	ReorderPoint	Name	ProductID
1	3	All-Purpose Bike Stand	879
2	3	AWC Logo Cap	712
3	3	Bike Wash - Dissolver	877
4	3	Cable Lock	843
5	3	Classic Vest, L	866
6	3	Classic Vest, M	865
7	3	Classic Vest, S	864

Birden fazla sıralama sütunu belirttiğimizde durum karışık gibi görülebilir. Aslında gayet basittir. İlk sıralanması istediğimiz sütunda aynı olan kayıtlar olduğu durumlarda, ikinci sıradaki sıralama sütununa göre sıralama işlemi gerçekleşir.

Yukarıdaki örneğimizde **ReorderPoint** sütunu değerinin aynı olması halinde, **Name** sütununa göre sıralama işlemi gerçekleşecektir.

Sıralama işlemi için sütun isimlerini yazmak mecburiyetinde değiliz. Bunun yerine yukarıda **SELECT** sorgusunda belirtilen sütunların sıra numarasını belirterek de bir sıralama gerçekleştirilebilir.

Sütun ismine göre hazırladığımız sorguyu sütun sırasına göre tekrar çalışıralım.

---

```
SELECT
    ReorderPoint, Name,
    ProductID
FROM
    Production.Product
ORDER BY
    3 ASC;
```

---

	ReorderPoint	Name	ProductID
1	750	Adjustable Race	1
2	750	Bearing Ball	2
3	600	BB Ball Bearing	3
4	600	Headset Ball Bearings	4
5	600	Blade	316
6	375	LL Crankarm	317
7	375	ML Crankarm	318

**Order By** 3, aslında **Order By ProductID** ile aynı anlama gelmektedir. 3 yerine 2 kullanılırsa, Name sütununa denk gelecektir ve sıralamayı Name sütununa göre gerçekleştirecektir.

**Order By** ile büyükten küçüğe ve küçükten büyüğe sıralama yapabileceğiniz rastgele sıralama da yapılabilir. Bu işlem için bir sistem fonksiyonu olan **NEWID()** kullanılabilir.

---

```
SELECT
    ProductID, Name
FROM
    Production.Product
ORDER BY
    NEWID();
```

---

	ProductID	Name
1	932	ML Road Tire
2	892	HL Touring Frame - Blue, 54
3	831	ML Mountain Frame - Black, 44
4	810	HL Mountain Handlebars
5	359	Thin-Jam Hex Nut 9
6	482	Metal Sheet 2
7	902	LL Touring Frame - Yellow, 58

## TOP OPERATÖRÜ

Yüksek miktarda kayıt içeren tablolarda belirli sayıda kayıt listelemek için kullanılır. SQL standartlarında yer alan bir komuttur. Listelenmesi istenen kayıtlar sayı olarak belirtilebileceği gibi dönen kayıtların yüzde ile oransal listelemesi de yapılabilir.

### Söz Dizimi:

---

```
SELECT TOP number|percent sutunlar
FROM tablo_isim
WHERE [condition]
```

---

**Production.Product** tablomuzdan dönen ilk 5 kaydı listeleyelim.

---

```
SELECT TOP 5 * FROM Production.Product;
```

---

	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

**TOP** operatörü ile yüzde kullanarak dönen kayıtların yüzde ile belirtilen kısmını da listeleyebiliriz.

---

```
SELECT TOP 1 PERCENT * FROM Production.Product;
```

---

	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

**TOP 1 PERCENT** bölümü, tüm ürünlerin %1'i kadarını listelemeyi belirtir.

Aynı soru yöntemi ile bir filtre de oluşturulabilir. Bu sayede istenilen kriterlerdeki kayıtlar arasında bir **TOP** işlemi gerçekleştirilir.

'c' ile başlayan ürünleri bularak dönen kayıtlardan 5 tanesini listeleyelim.

---

```
SELECT TOP 5 * FROM Production.Product WHERE Name LIKE 'C%';
```

---

	ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	320	Chainring Bolts	CB-2903	0	0	Silver	1000	750	0,00	0,00
2	321	Chainring Nut	CN-6137	0	0	Silver	1000	750	0,00	0,00
3	322	Chainring	CR-7833	0	0	Black	1000	750	0,00	0,00
4	323	Crown Race	CR-9981	0	0	NULL	1000	750	0,00	0,00
5	324	Chain Stays	CS-2812	1	0	NULL	1000	750	0,00	0,00

## TOP FONKSİYONU

T-SQL fonksiyonlarından biri olan **TOP** fonksiyonu, **TOP** operatörü ile benzer şekilde çalışır. **Order By** ile birlikte kullanılır.

### Söz Dizimi:

---

```
TOP (expression) PERCENT WITH TIES
```

---

**Production.Product** tablosundaki kayıtların ilk 5 tanesini, **ProductID** sütununa göre listeleyerek sıralayalım.

---

```
SELECT
    TOP(5) WITH TIES *
FROM
    Production.Product
ORDER BY
    ProductID;
```

---

	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

**Production.Product** tablosundaki kayıtların ilk %5'lik kısmını **ProductID** sütununa göre listeleyelim.

---

```
SELECT
    TOP(5) PERCENT WITH TIES *
FROM
    Production.Product
ORDER BY
    ProductID;
```

---

	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

Bu sorguda bize yüzde özelliğini kazandıran **PERCENT** komutudur.

## TABLOLARI BİRLEŞTİRMEK

İlişkisel verilerde farklı tablolarda yer alan ve birbirini tamamlayan kayıtların tek bir tabloymuş gibi sorgu sonucunda birleştirerek listeleme işlemleri **JOIN** ifadeleri ile gerçekleştirilebilir.

Bir tablo kendisi ile birleştirilebileceği gibi farklı birden fazla tablo ile de birleştirilebilir. Birleştirme işleminde istenen işleme göre farklı yaklaşımlar mevcuttur. Bu yaklaşımların tamamını karşılamak için **JOIN**'ler kendi içerisinde farklı isimlendirmeler ve farklı komutlara sahiptir.

Örneğin; e-ticaret sisteminde, müşterilerin siparişlerini tutan tablo ile müşteri tablosu birleştirilerek müşterinin hangi ürünlerin sipariş ettiğini, hatta hangi kategorilerdeki ürünleri karşı daha fazla satın alma işlemini gerçekleştirdiği gibi bilgileri listelemek için kullanılabilir.

İleri seviye veritabanı programlama için **JOIN**'ler vazgeçilmez ve önem arz eden konulardan biridir. Genel olarak veritabanı programlamada kavram açısından karıştırılan bir konu olması nedeni ile bu konuya geniş yer ayırarak detaylı örnek ve analizler gerçekleştireceğiz.

**JOIN**'ler, her ne kadar karmaşık yapılara sahip olabilse de temel söz dizimi şu şekildedir.

---

```
SELECT sutun_ismi1[, sutun_ismi2, ...]
FROM tablo_ismi1 [, tablo_ismi2, ...]
WHERE tablo_ismi1.sutun_ismi1 = tablo_ismi2.sutun_ismi2
...
```

---

Bu **JOIN** kullanımı, klasik **JOIN** modelidir. Karmaşık olmayan, ancak ileri seviye uygulamalarda çok kullanılmayan temel bir **JOIN** yapısıdır. **ANSI** standartlarında bir **JOIN** söz dizimi olduğu için SQL Server'da desteklemektedir.

## KLASİK JOIN

**WHERE** cümleciği ile gerçekleştirilen bu **JOIN**, **ANSI** standartlarında ve yukarıda belirttiğimiz söz dizimine sahip olan **JOIN** yöntemidir. SQL Server klasik **JOIN**'ı bir **INNER JOIN** olarak işleme alır.

Satılan ürünlerini listeleyelim.

---

```
SELECT SOD.ProductID, P.Name
FROM Sales.SalesOrderDetail AS SOD, Production.Product AS P
WHERE SOD.ProductID = P.ProductID;
```

---

	ProductID	Name
1	707	Sport-100 Helmet, Red
2	707	Sport-100 Helmet, Red
3	707	Sport-100 Helmet, Red
4	707	Sport-100 Helmet, Red
5	707	Sport-100 Helmet, Red
6	707	Sport-100 Helmet, Red
7	707	Sport-100 Helmet, Red

Bu sorgu ile satılan tüm ürünler listelenir. Ürün birden fazla satılmış da olsa satıldığı kadar kayıt listlenecektir.

Geliştirilmesi en kolay **JOIN** yöntemidir. Temel kullanımı kavrayabilmek için birleştirmek istediğiniz iki tabloyu da inceleyebilirsiniz.

Örneğin, yukarıdaki sorguda **Sales.SalesOrderDetail** ve **Production.Product** tabloları kullanıldı. Bu işlemde belirlememiz gereken öncelik hangi sütunların birleştirileceğidir.

---

İki tabloyu da alt alta tek seferde çalıştırarak tablo yapısını inceleyebilirsiniz.

---

```
SELECT * FROM Sales.SalesOrderDetail
SELECT * FROM Production.Product
```

---

SalesOrderID	SalesOrderDetailID	CarrierTrackingNumber	OrderQty	ProductID	SpecialOfferID	UnitPrice	UnitPriceDiscount
1	43659	1	4911-403C-98	1	776	1	2024,994 0,00
2	43659	2	4911-403C-98	3	777	1	2024,994 0,00
3	43659	3	4911-403C-98	1	778	1	2024,994 0,00

ProductID	Name	ProductNumber	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	Standard
1	Adjustable Race	AR-5381	0	0	NULL	1000	750	0,00
2	Bearing Ball	BA-8327	0	0	NULL	1000	750	0,00
3	BB Ball Bearing	BE-2349	1	0	NULL	800	600	0,00
4	Headset Ball B...	BE-2908	0	0	NULL	800	600	0,00
5	Blade	BL-2036	1	0	NULL	800	600	0,00

Bu sorgu sonucunda iki tablodaki verilerde alt alta listelenecektir. Birleştirmek istenen sütunlar belirlenerek sorgu şu şekilde yazılabilir.

- Birleştirmek istenen sütunlar belirlenir.

`Sales.SalesOrderDetail` tablosundaki `ProductID` sütunu ile `Production.Product` içerisindeki `Name` sütunu birleştirilecek.

Yani aşağıdaki gibi, şema\_ismi.tablo\_ismi.sütun\_ismi olarak nitelendirebiliriz.

---

`Sales.SalesOrderDetail.ProductID - Production.Product.Name`

---

- Tablo isimlendirmesi uzun olacağı için basit bir sorgu bile çok karmaşık hal alabilir. Bu nedenle tabloları takma ad ile isimlendirin.

...  
`FROM Sales.SalesOrderDetail AS SOD, Production.Product AS P`  
 ...

---

Artık bir önceki uzun isimlendirme şu şekilde olacaktır.

---

`SOD.ProductID - P.Name`

---

- Son olarak belirlenen ve kısaltılan isimler ile birlikte WHERE cümlesi içinde tabloları eşitleyin.

...  
`WHERE SOD.ProductID = P.ProductID`  
 ...

---

`JOIN` işleminin temel mantığı bundan ibarettir. `JOIN` işlemini zorlaştıran şey sorgu yapısı değil, birleştirilecek tablolardan çokluğu ve listelenmek istenen verinin karmaşık özelliklere sahip olmasıdır.

## SQL SERVER'DA JOIN MİMARİSİ

SQL Server performans ve sonuç olarak en iyi işlemi gerçekleştirecek `JOIN`'ı kendisi belirler.

SQL Server, `JOIN` sorgularını oluşturmak için şu alt seviye `JOIN` yöntemlerini kullanır.

- **Merge Join:** Birinci tablodan bir kayda karşılık ikinci tablodan alınan kaydın ilişkili olması şartı ile çalışır.
- **Hash Join:** Tablolardan her ikisinde de uygun bir indeks bulunamazsa ya da WHERE cümleciği yer alıyorsa kullanılır.
- **Nested Loop Join:** Birleştirilecek iki tablodan az kaydı olan tablo dışarıdaki, çok olan tablo içerisindeki döngü gibi çalışır. Adından da anlaşılacağı gibi içe iki döngü gibi çalışır.

SQL Server'da **JOIN** işlemlerini bu 3 yöntemden birine göre çalışırmak için özel ayar belirtilebilir.

---

Ayar belirtme işlemi için **JOIN** sorgusunun altına şu şekilde bildirim yapılır.

---

OPTION (MERGE JOIN)

---

### Örnek:

---

```
SELECT S.CountryRegionCode, S.StateProvinceCode,
       T.TaxType, T.TaxRate
  FROM Person.StateProvince AS S
 LEFT OUTER JOIN Sales.SalesTaxRate AS T
    ON S.StateProvinceID = T.StateProvinceID
OPTION (MERGE JOIN)
```

---

	CountryRegionCode	StateProvinceCode	TaxType	TaxRate
1	CA	AB	1	14,00
2	CA	AB	2	7,00
3	US	AK	NULL	NULL
4	US	AL	NULL	NULL
5	US	AR	NULL	NULL
6	AS	AS	NULL	NULL
7	US	AZ	1	7,75

Diğer yöntemlerde aynı şekilde kullanılabilir.

---

OPTION (LOOP JOIN)

---

ya da

---

OPTION (HASH JOIN)

---

## INNER JOIN

İki tablo arasında birleştirme yaparken, her iki tabloda da yer alan değerler seçilir. Tablolardan sadece birinde yer alıp, diğerinde ilişkili değer bulunmayan kayıtlar seçilmmez. **JOIN** yöntemlerinde en çok kullanılan tablo birleştirme yöntemidir.

Personellerin ad, soyad, personel türü ve telefon numarası bilgilerini listeleyelim.

---

```
SELECT P.FirstName, P.LastName, P.PersonType, PP.PhoneNumber
FROM Person.Person AS P
INNER JOIN Person.PersonPhone AS PP
ON P.BusinessEntityID = PP.BusinessEntityID;
```

---

	FirstName	LastName	PersonType	PhoneNumber
1	Ken	Sánchez	EM	697-555-0142
2	Teri	Duffy	EM	819-555-0175
3	Roberto	Tamburello	EM	212-555-0187
4	Rob	Walters	EM	612-555-0100
5	Gail	Erickson	EM	849-555-0139
6	Jossef	Goldberg	EM	122-555-0189
7	Dylan	Miller	EM	181-555-0156

Bu tablo birleştirme işleminde **Person.Person** ve **Person.PersonPhone** tablolarında ortak sütun olan **BusinessEntityID** sütunları üzerinden birleştirme işlemi gerçekleştirerek, her iki sütunduda da birbiri ile eşleşen kayıtları listeledik. İki sütunu da birleştirdiğimiz için, her iki sütundan da istediğimiz sütunları takma isimlerini kullanarak sonuç listemize ekledik.

**JOIN** işleminde bir ya da birden fazla tablo birleştirilebilir. SQL Server aynı anda 256 tabloyu **JOIN** ile birleştirebilir.

Genel kullanılan **JOIN** türü **INNER JOIN** olsa da, bazı soru örneklerinde, sadece **JOIN** komutunun kullanıldığını görmüş olabilirsiniz. **INNER** yazılmadan, sadece **JOIN** kullanılan birleştirme işlemlerinde SQL Server, **INNER JOIN** işlemi gerçekleştirir. Yani **JOIN** ile **INNER JOIN** aynı yöntem ile çalışır.

## OUTER JOIN

Aynı anda her iki tabloda da yer almayan kayıtları listelemek için kullanılır.

**OUTER JOIN** kendi içerisinde farklı özellikler için kullanılan 3 yan özelliğe sahiptir. Bu kullanımlar ile iki tablodan biri ya da her ikisi üzerinde bu işlemi gerçekleştirebilir.

**OUTER JOIN** sorgusunda kullanılan ilk tablo soldaki tablo, ikinci tablo ise sağdaki tablo olarak kabul edilir.

### OUTER JOIN TİPLERİ

- **LEFT:** Soldaki tabloda yer alan kayıtlar, sağdaki tabloda karşılıkları olmasa bile getirilirler.
- **RIGHT:** Sağdaki tabloda yer alan kayıtlar, soldaki tabloda karşılıkları olmasa bile getirilirler.
- **FULL :** Soldaki ve sağdaki tabloda karşılıklı olarak eşit satırı olmayan kayıtlar getirilir.

### LEFT OUTER JOIN

Birleştirilen iki tablodan soldaki (birinci tablo) tabloda bulunan tüm kayıtlar getirilir. Sağdaki (ikinci tablo) tabloda ise soldaki tablo ile ilişkili ve eşleşen kayıtlar getirilir.

---

```
SELECT S.CountryRegionCode, S.StateProvinceCode,
       T.TaxType, T.TaxRate
  FROM Person.StateProvince AS S
 LEFT OUTER JOIN Sales.SalesTaxRate AS T
    ON S.StateProvinceID = T.StateProvinceID;
```

---

	CountryRegionCode	StateProvinceCode	TaxType	TaxRate
1	CA	AB	1	14,00
2	CA	AB	2	7,00
3	US	AK	NULL	NULL
4	US	AL	NULL	NULL
5	US	AR	NULL	NULL
6	AS	AS	NULL	NULL
7	US	AZ	1	7,75

`LEFT OUTER JOIN` ile gerçekleştirdiğimiz bu sorguyu `INNER JOIN` ile hazırlasaydık, `TaxType` ve `TaxRate` sütunlarında `NULL` olan kayıtlar listelenmeyecekti.

Yukarıdaki ile aynı sorguyu, sadece `JOIN` kısmını değiştirerek `INNER JOIN`'e dönüştürebiliriz.

---

```
SELECT S.CountryRegionCode, S.StateProvinceCode,
       T.TaxType, T.TaxRate
  FROM Person.StateProvince AS S
 INNER JOIN Sales.SalesTaxRate AS T
    ON S.StateProvinceID = T.StateProvinceID;
```

---

	CountryRegionCode	StateProvinceCode	TaxType	TaxRate
1	CA	AB	1	14,00
2	CA	ON	1	14,25
3	CA	QC	1	14,25
4	CA	AB	2	7,00
5	CA	ON	2	7,00
6	CA	QC	2	7,00
7	CA	BC	3	7,00

`LEFT OUTER JOIN` işlemlerinde `OUTER` anahtar kelimesini zorunlu değildir. Sadece `LEFT JOIN` yazarak da kullanılabilir.

## RIGHT OUTER JOIN

Birleştirilen iki tablodan sağdaki (ikinci tablo) tabloda bulunan tüm kayıtlar getirilir. Soldaki (birinci tablo) tabloda ise sağdaki tablo ile ilişkili ve eşleşen kayıtlar getirilir.

`LEFT JOIN` işleminde kullandığımız sorguda ü sadece `LEFT` kısmını `RIGHT` ile değiştirerek çalıştıralım.

---

```
SELECT S.CountryRegionCode, S.StateProvinceCode,
       T.TaxType, T.TaxRate
  FROM Person.StateProvince AS S
 RIGHT OUTER JOIN Sales.SalesTaxRate AS T
    ON S.StateProvinceID = T.StateProvinceID;
```

---

	CountryRegionCode	StateProvinceCode	TaxType	TaxRate
1	CA	AB	1	14,00
2	CA	ON	1	14,25
3	CA	QC	1	14,25
4	CA	AB	2	7,00
5	CA	ON	2	7,00
6	CA	QC	2	7,00
7	CA	BC	3	7,00

Sonuç olarak 29 kayıt listelenmiştir. Fark ettiyseniz **LEFT JOIN** işleminden sonra kullandığımız **INNER JOIN** sorgusu ile aynı sonucu getirmiştir.

## FULL OUTER JOIN

İki tablo üzerinde hem **RIGHT JOIN** hem de **LEFT JOIN** işlemi gerçekleştirir.

İlk olarak soldaki tabloda bulunan tüm kayıtlar listelenir ve bu kayıtların sağdaki tabloda karşılıkları varsa, onlar listelenir ve karşılığı olmayanlar **NULL** olarak listelenir. Daha sonra sadece sağdaki tabloda olup, soldaki tabloda olmayan kayıtlarda listelenir.

---

```
SELECT S.CountryRegionCode, S.StateProvinceCode,
       T.TaxType, T.TaxRate
  FROM Person.StateProvince AS S
 FULL OUTER JOIN Sales.SalesTaxRate AS T
    ON S.StateProvinceID = T.StateProvinceID;
```

---

	CountryRegionCode	StateProvinceCode	TaxType	TaxRate
1	CA	AB	1	14,00
2	CA	AB	2	7,00
3	US	AK	NULL	NULL
4	US	AL	NULL	NULL
5	US	AR	NULL	NULL
6	AS	AS	NULL	NULL
7	US	AZ	1	7,75

**FULL OUTER JOIN** kullanırken **OUTER** yazılmak zorunda değildir. Sadece **FULL JOIN** olarak kullanılabilir.

## CROSS JOIN

İki tabloda yer alan kayıtları çaprazlamak için kullanılır. Bu işleme matematiksel olarak kartezyen çarpımı denir. İki tablo arasında herhangi bir ilişkilendirme olmasına gerek yoktur.

5 kayıt içeren A tablosu ile 3 kayıt içeren B isimli iki tabloda **CROSS JOIN** işlemi uygulandığında, sonuç listesi  $5 \times 3$  kayıtтан oluşacaktır.

---

```
SELECT p.BusinessEntityID, t.Name AS Territory
FROM Sales.SalesPerson p
CROSS JOIN Sales.SalesTerritory t
ORDER BY p.BusinessEntityID;
```

---

	BusinessEntityID	Territory
1	274	Northwest
2	274	Northeast
3	274	Central
4	274	Southwest
5	274	Southeast
6	274	Canada
7	274	France

Sorgu sonucunda 170 kayıt listlenecektir. **CROSS JOIN** işlemi uygulanan iki tablonun toplam kayıt sayılarına bakalım.

---

```
SELECT COUNT(BusinessEntityID) FROM Sales.SalesPerson;
```

---

	(No column name)
1	17

---

```
SELECT COUNT(TerritoryID) FROM Sales.SalesTerritory;
```

---

	(No column name)
1	10

$$17 \times 10 = 170$$

**CROSS JOIN**, birinci tabloda yer alan her bir kaydı ikinci tabloda yer alan her bir kayıt ile ilişkilendirerek satırlar türetmede kullanılır. **CROSS JOIN** sıklıkla kullanılan bir **JOIN** işlemi değildir. Ancak bazı satır türetme isteklerinde kullanılabilir.



# 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 koparılmamasını 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 tutarlığını sağlamak için gerekli özellikler 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 etkilisi **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 oluşturulur.

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 seçilmemesine 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 satırındaki sütunun **unique** (*benzersiz*) değerlere sahip olmasını sağlamak buna bir örnek olabilir.

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

- **Referential Integrity Constraint**

Bir sütundaki değerin, 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 birbirile ilişkili olma durumunu gerçekleştirir. Kategoriler tablosunda 3 no'lu kategori yok ise, **Makaleler** tablosundaki **KategoriID** sütununda 3 değerinin bulunması ne kadar mantıklı olabilir? Bu ilişkisel veri bütünlüğünü sağlamak için **Referential Integrity Constraint** kullanılır. Bu constraint'lere örnek olarak **Foreign Key Constraint** verilebilir.

## CONSTRAINT İSİMLENDİRMEŞİ

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. Sonrasında 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 oluşturulursaydı, 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 sağlanması 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 oluşturulurken, 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)	(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(
    Kullanicid INT PRIMARY KEY NOT NULL,
    Ad VARCHAR(50),
    Soyad VARCHAR(50),
    KullaniciAd VARCHAR(20)
);
```

---

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

---

```
ALTER TABLE Kullanicilar
ADD CONSTRAINT PKC_KullaniciID PRIMARY KEY(KullaniciID);
```

---

Bu işleme, **KullaniciID** 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)	PersonellID
2	UNIQUE (non-clustered)	UQ_Personel_E0103670690944F2	(n/a)	(n/a)	(n/a)	(n/a)	KullaniciAd

## MEVÇUT 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 gelecek bir değer tanımlama işlemi için kullanılır.

Örneğin; kullanıcılar tablosunda kullanıcının kayıt olduğu zaman, bilgisi istemci tarafından parametre olarak bildirilmek zorunda değildir. SQL Server tarafından, 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 sorusu 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,'SamilAyyildiz','samil.ayyildiz@abc.com', 'İstanbul');
```

---

Eklenen veriyi listeleyelim.

---

```
SELECT * FROM Personeller;
```

---

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

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

Oluşturulan yeni **DEFAULT** constraint'i 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 KayitTarih	DF__Personell__Kayit__2B5F6B28	(n/a)	(n/a)	(n/a)	(n/a)	(getdate())
2	PRIMARY KEY (clustered)	PK__Personel__0F0C57518AA38D7D	(n/a)	(n/a)	(n/a)	(n/a)	PersonelID

## VAR OLAN TABLOYA DEFAULT CONSTRAINT EKLEMELİ

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 KayitTarih
```

---

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 işlev 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 **Kullanıcılar** isimli yeni bir tablo oluşturalım.

---

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

---

**Tablo, key ve nesneler** arası ilişkilerin tasarımları 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 oluşturmazsa, 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 yapılsa hata üretecektir.

Geliştirilen uygulama her yönüyle hesaplanmalıdır. Kullanıcının en az kaç karakter girmesi istediği, en fazla kaç karakter girmesini istediği, 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 @ işaretinin 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 @ işaretini 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 @ işaretini 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','sifrel','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.

İstedigimiz telefon formatı şu: **0XXXXXXXXXX**

---

```
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) = 11)
```

---

Telefonun ilk karakterinin 0 (*sıfır*) olmasını ve kalan karakterlerinde doğal olarak nümerik tekli sayılar arasında (0-9) olmasını şart koşuyuk. 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 sorğu 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][0-9]' )
AND LEN(Telefon) = 14)
```

---

3 özel karakteri de ekleyerek toplam 14 karakterlik bir veri girişi yapılması gereklidir. Bu durumda tabii 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 giriş 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 kaydolan 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 kırılamaz 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),
    EklemeTarih     DATETIME
);
```

---

Referans alınacak tablodaki gerçek sütunda **PRIMARY KEY** ya da **UNIQUE** Constraint tanımlanmış olması gereklidir. 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)
```

---

İşlemin doğruluğunu test etmek için her zaman tablo üzerindeki constraint'leri görüntülemeyi ihmal etmeyin.

---

```
sp_helpconstraint 'Makaleler'
```

---

	Object Name	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	Makaleler	FOREIGN KEY	FK_Makaleler_Kateg_02284B6B	No Action	No Action	Enabled	Is_For_Replication	KategoriID REFERENCES AdventureWorks2012...
2		FOREIGN KEY	FK_MakaleKategoriler	No Action	No Action	Enabled	Is_For_Replication	KategoriID REFERENCES AdventureWorks2012...
3		PRIMARY KEY (clustered)	PK_Makalele_DA97CBAD98ABB90B	(n/a)	(n/a)	(n/a)	(n/a)	MakaleID
4								
5								

## RULE VE DEFAULT

Rule ve Default'lar Check ve Default constraint'lerin görevlerini yerine getirebilirler. SQL Server'in eski sürümlerinde kullanılan ve geriye doğru uyumluluk adına halen desteklenen, ancak sonraki SQL Server versiyonlarında desteğin kalkacağını bildiğimiz nesnelerdir. Check ve Default Constraint'leri Rule ve Default nesnelerine göre daha performanslıdır. Rule ve Default nesneleri özel birer nesnedir. Tablodan bağımsızdır ve özel olarak derlenerek oluşturulur ve daha sonra tabloya ilişkilendirilirler. Constraint'ler ise, tablo özellikleri olarak kullanılır.

## CONSTRAINT'LERİ İNCELEMEMEK

Bir tablodaki var olan constraint'leri listelemek gerekebilir. Constraint tipi, ismi, durumu gibi bilgileri öğrenebilmek için `sp_helpconstraint` kullanılır.

Production.Product tablosundaki constraint'leri listeleyelim.

---

```
EXEC sp_helpconstraint 'Production.Product';
```

---

	Object Name	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	Production.Product	CHECK on column Class	CK_Product_Class	(n/a)	(n/a)	Enabled	Is_For_Replication	(upper([Class])='H' OR upper([Class])='M' OR upp...
2		CHECK on column DaysToManufacture	CK_Product_DaysToManufacture	(n/a)	(n/a)	Enabled	Is_For_Replication	((DaysToManufacture)>=0)
3		CHECK on column ListPrice	CK_Product_ListPrice	(n/a)	(n/a)	Enabled	Is_For_Replication	((ListPrice)>=0.00)
4		CHECK on column ProductLine	CK_Product_ProductLine	(n/a)	(n/a)	Enabled	Is_For_Replication	((upper([ProductLine])='R' OR upper([ProductLine]...))
5		CHECK on column ReorderPoint	CK_Product_ReorderPoint	(n/a)	(n/a)	Enabled	Is_For_Replication	((ReorderPoint)>0)
6		CHECK on column SafetyStockLevel	CK_Product_SafetyStockLevel	(n/a)	(n/a)	Enabled	Is_For_Replication	((SafetyStockLevel)>0)
7		CHECK Table Level	CK_Product_SellEndDate	(n/a)	(n/a)	Enabled	Is_For_Replication	((SellEndDate)>=(SellStartDate) OR [SellEndDat...
		Table is referenced by foreign key						
1	AdventureWorks2012.Production.BillOfMaterials_F...							
2	AdventureWorks2012.Production.BillOfMaterials_F...							
3	AdventureWorks2012.Production.ProductCostList...							
4	AdventureWorks2012.Production.ProductDocumen...							
5	AdventureWorks2012.Production.ProductInventory...							
6	AdventureWorks2012.Production.ProductListPrice...							
7	AdventureWorks2012.Production.ProductProductP...							

## **CONSTRAINT'LERİ DEVRE DİŞİ BIRAKMAK**

Bazen constraint ile gerçekleştirilen kontroller, yani constraint'ler bir süreliğine ya da tamamen kaldırırmak 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 gerçekleştirilmeydi. 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 işlemden sonra gerçekleştirilecek veri eklemelerinde, artık bu telefon formatına uyulmak zorundadır.

## CONSTRAINT'İ GEÇİCİ OLARAK DEVRE DİŞİ BIRAKMAK

SQL Server'in 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'in bir süreliğine bu duruma göz yummamasını sağlayabiliyoruz.

**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.



# İLERİ SEVİYE SORGULAMA

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 NEDİR?

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 içe 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 döndürmek 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)
```

---

## TEKİL 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 belirtildesinden 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, istedigimiz 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 kullanılabılır.

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 anlaşılabılır 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 soruyu 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 soru 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 soru 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 gerekiği durumlar elbette vardır. Ancak **JOIN** ile gerçekleştirebileceğiniz sorgularda **JOIN**'ı tercih etmenizi öneririm. Performans açısından bazı durumlar haricinde **JOIN** daha etkilidir.

## TÜRETİLMİŞ 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 içerisinde 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
```

---

## **İLİŞKİLİ ALT SORGULAR**

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

### **İLİŞKİLİ ALT SORGULAR NASIL ÇALIŞIR?**

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 LİSTESİNDEKİ İLİŞKİLİ ALT SORGULAR**

Ürünler tablosundaki alt kategori ile alt kategoriler tablosundaki alt kategorileri ilişkilendirerek, her ürünün hangi alt kategoride yer aldığıni 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 KOŞULUNDAKİ İLİŞKİLİ 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 soru 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 soru üzerinde **EXIST**'ı 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 dönmese, dışarıdaki sorgu çalıştırılmaz ve **FALSE** değeri üretir.

## VERİ TIPLERİNİ 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 işlevi 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] )
```

---

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

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

---

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

---

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

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 olmalıdır. Öztle, 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
2	679	Silver
3	680	Rear Derailleur Cage
4	706	Black
5	707	HL Road Frame - Red, 58
6	708	Red
7	709	Sport-100 Helmet, Red
		Black
		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 **ÜrünAd’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 soru içerisinde kullanılabilir.

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

---

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

---

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

## RÜTBELEME FONKSİYONLARI İLE KAYITLARI SIRALAMAK

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	Adjustable Race	0,00
2	2	Bearing Ball	0,00
3	3	BB Ball Bearing	0,00
4	4	Headset Ball Bearings	0,00
5	316	Blade	0,00
6	317	LL Crankarm	0,00
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 FONKSİYONLARI

**RANK** ve **DENSE\_RANK** fonksiyonları kayıtları listelerken her bir kayıt için bir sıra numarası vermeye yarar.

### RANK

**RANK** fonksiyonu, bir veri kümesi içinde graplama yaparak her bir veriyi belirtilen kriteri 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 listeleneciktir. 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 OPERATÖRLERİ

**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 cümlesi 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 özelliklerini 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 işlev sahiptir. **PIVOT** satırları sütunlara dönüştürken **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 içerisindeki 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 yaptığına 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 kesşim işlemeye tabi tuttuğumuzda, iki tabloda kesisen 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
2
3
4
5
6
7
863
862
861
860
859
858
878

## **TRUNCATE TABLE ile Veri Silmek**

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 tasarılandı 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;
```

---

# İLERİ VERİ YÖNETİM TEKNİKLERİ

## VERİ 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 YENİ 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 istediğimiz 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.

## VERİ GÜNCELLEME

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

### TABLOLARI BİRLEŞTİREREK VERİ 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 VERİ GÜNCELLEMEMEK

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 VERİLERİ GÜNCELLEMEMEK

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

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

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 eğitim 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	Online eğitim sistemi

## VERİ 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 BİRLEŞTİREREK VERİ SİLMEK

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 soru 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 soru 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 soru sonucunda, **JobID** değeri 2 olan kayıtlar silinecektir.

Ancak bu soruyu 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 soru 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 VERİ SİLMEK**

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 FONKSİYONU İLE VERİ SİLMEK**

**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 değişmekte birlikte, tablo üzerinde daha önce silme işlemi gerçekleştirmediyiseniz, 27 kayıt silinecektir.

Sayı ile kayıt silmek için;

---

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

---

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

## SİLİNEN BİR KAYDIN DELETED İÇERİSİNDE GÖRÜNTÜLENMESİ

Silinen bir kaydı, **silindi** anda, ekranda görüntülemek için;

---

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

---

## DOSYALARIN VERİTABANINA EKLENMESİ VE GÜNCELLENMESİ

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	0x0000018667479706D703432000000069736F6D6D703432000114926D6F6F760000006C6D76686400000000CC327FE1CC327FE100002580001FCB60...

Bu sorgu çalıştırıldığında, veritabanında bulunmayan, **C:\** dizinindeki **video.mp4** dosyasının Binary formatındaki değerine ulaşılır. Bu dönüştürme işlemini gerçekleştiren, tabii 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	0x0000001C66747970464143450000053969736F6D6176633146414345000109AE6D6F6F760000006C6D7668640000000...

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	0x0000001C66747970464143450000053969736F6D6176633146414345000109AE6D6F6F760000006C6D7668640000000...
2	1	0x89504E470D0A1A0A000000D49484452000007200000030080600001BA4A1E8C00000097048597300000...

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

Binary olarak eklediğimiz dosya kayıtlarının üzerinde güncelleme işlemi de gerçekleştirilebiliriz.

*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 faydalarnı 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 bağı 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 şışmesinin 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 özellikleir. **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 Özüğünü Aktifleştirmek

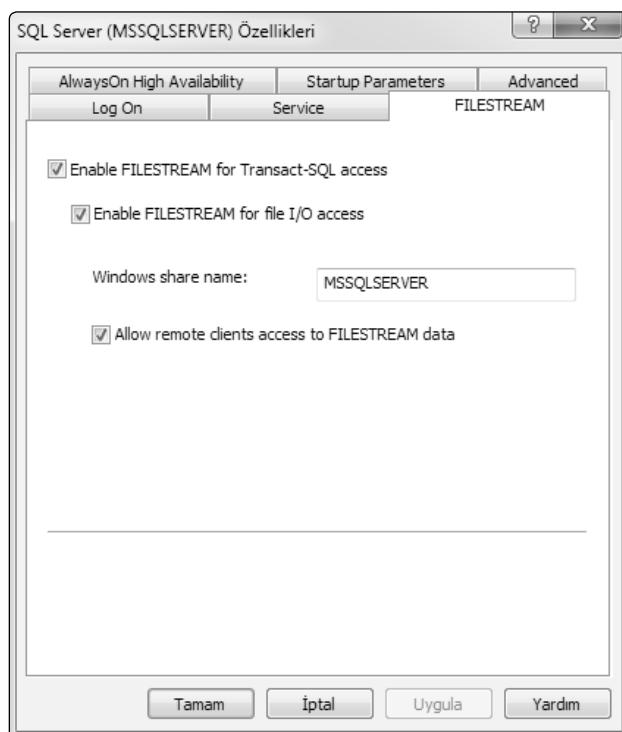
**FILESTREAM** özelliği; SQL Server kurulumunda varsayılan (*default*) olarak kapalı (*pasif*)dır. Bu özelliği kurulum sırasında değiştirmediyerseniz daha sonra da değiştirebilirsiniz.

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

- SQL Server Configuration Manager arayüzüünü açmak için;

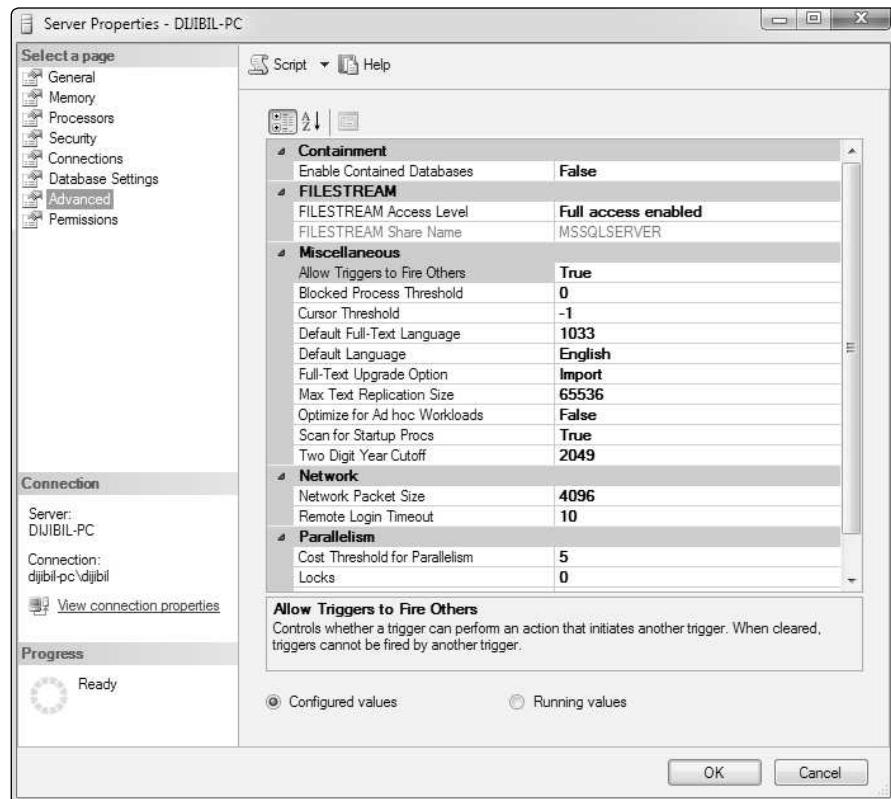
*Programlar\Microsoft SQL Server 2012\Configuration Tools\SQL Server Configuration Manager*

**Configuration Manager** aracında **FILESTREAM** ayarlarını gerçekleştirmek için, sol bölümdeki **SQL Server Services** seçilmeli, daha sonra sağ kısmda **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 BİR VERİTABANINDA 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 karşılaşmanız için; Binary edilecek dosyayı belirttiğiniz **SELECT** sorgusundan sonra, başka bir parametre daha ekleyecekseniz, **SELECT** sorgusunu, takma ismi ile birlikte parantez içerisinde almalısınız. Aksi halde sorgu hata üretecektir.

## FILESTREAM ÖZELLİĞİ AKTİF EDİLMİŞ BİR VERİTABANI 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 istediğimiz 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.



## SÜTUNLARI OLUŞTURMAK

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

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

---

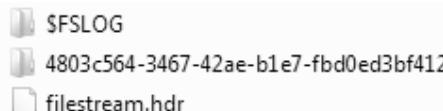
```
CREATE TABLE Person(
    PersonID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    Email   VARCHAR(50),
    PImage VARBINARY(MAX) FILESTREAM NULL
);
```

---

**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 oluşturuktan 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.

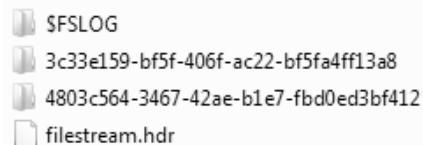
---

```
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.



## VERİ 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;
```

---

## VERİ SEÇMEK

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

Person tablosuna eklediğimiz kaydı listeleyelim.

---

```
SELECT * FROM Person;
```

---

## VERİ GÜNCELLEMEMEK

**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 güncelllemeye 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 belirtmiştık. 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** sorgusunu çalıştırın ve listelenen sonuçlardan **PersonID** üzerinde ilgili kaydın hücresiné sağ tıklayarak kopyalayın.

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

## GUID DEĞERE SAHİP SÜTUN OLUŞTURULURKEN DİKKAT EDİLMESİ 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 gerekiyordu. **GUID** olarak kullanılacak sütunu ise, ayrı bir sütun olarak oluşturulmalıdır.

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.

## VERİ SİLMEK

**FILESTREAM** tablolarda veri silme işlemi T-SQL tarafından 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.

## VERİLERİ 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, grplara bölmek için kullanılır. Genel olarak grup başına ayrı istatistikler üretirmek, 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 istediğimizi karşılayacak işlem verileri gruplamak ile gerçekleştirilebilir. Bunun için **GROUP BY** komutunu kullanarak bu sorguyu tekrar çalışıralım.

```
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 böülümlendirmek 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 graplama yapılmış veri üzerinde de `order By` kullanarak listeleme işlemi yapabilirsiniz.

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



Aggregate konusu **Graplamlı Fonksiyonlar** bölümünde detaylıca inceleneciktir.

## GROUP BY ALL

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

Bu iki graplama 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 [ToplamSatilanUrun]
FROM
Sales.SalesOrderDetail
WHERE ProductID < 800
GROUP BY ProductID;
```

	ProductID	ToplamSatilanUrun
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 listeleneciktir.

Şimdi aynı soruyu **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_ismi, Gruplamali_Fonksiyon(sutun_ismi)
FROM tablo_ismi
WHERE şartlar
GROUP BY sutun_ismi
HAVING Gruplamali_Fonksiyon(sutun_ismi)
[ORDER BY sıralayıcı]
```

---

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

## GRUPLAMALI FONKSİYONLAR (AGGREGATE FUNCTIONS)

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

### AVG FONKSİYONU

Average'in 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 çalıştığımız `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 Crankam	0,00
7	ML Crankam	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 FONKSİYONU

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 ayrılmadan 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 çalıştmak 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 FONKSİYONU

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	(No column name)
FROM Production.Product;		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	(No column name)
FROM Production.Product;		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

Göründüğü 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 (\*) işaretini herhangi bir sütununda değer olan tüm girilmiş kayıtları toplar. Sütun bazlı **COUNT(sütun\_ad)** kullanım gerçekleştirilirse **NULL** kayıtlar hesap dışı tutulacaktır. Ancak **COUNT(\*)** işlemine tabi tutulan tablodaki **NULL** olanlar dahil tüm kayıtlar hesaplanacaktır.

## MAX FONKSİYONU

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

### Söz Dizimi:

---

```
SELECT MAX(sütun_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 gerçekleştiriyorsa, `ASCENDING(ASC)` işleminde nasıl A'dan başlayan bir sıralama işlemi gerçekleştiriyor 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;
```

Göründüğü 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 FONKSİYONU

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
FROM
    Production.Product
WHERE
    ProductID = (SELECT
        MIN(ProductID)
        FROM Production.Product);
```

76.PNG

	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.

## **GRUPLANMIŞ VERİLERİ 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 listeleneciktir.

---

```
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	İleri Seviye Java Programlama	8
2	Software	PHP	PHP 6	6
3	Software	PHP	PHP 6 ile E-Ticaret Uygulaması Geliştirme	6
4	Software	Android	İleri Seviye Android Programlama	9
5	Database	SQL Server	İleri Seviye SQL Server T-SQL	9
6	Database	SQL Server	SQL Server Veritabanı Yönetimi	9
7	Database	SQL Server	SQL Server 2012 Yenilikleri	7
8	Database	Oracle	İleri Seviye Oracle PL/SQL	9
9	Database	Oracle	Oracle PL/SQL	9
10	Database	Oracle	Oracle Veritabanı Yönetimi	9
11	Database	Oracle	Oracle İş Uygulamaları	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	İleri Seviye Oracle PL/SQL	9
2	Database	Oracle	Oracle İş Uygulamaları	9
3	Database	Oracle	Oracle PL/SQL	9
4	Database	Oracle	Oracle Veritabanı Yönetimi	9
5	Database	Oracle	NULL	36
6	Database	SQL Server	İleri Seviye SQL Server T-SQL	9
7	Database	SQL Server	SQL Server 2012 Yenilikleri	7
8	Database	SQL Server	SQL Server Veritabanı Yönetimi	9
9	Database	SQL Server	NULL	25
10	Database	NULL	NULL	61
11	Software	Android	İleri Seviye Android Programlama	9
12	Software	Android	NULL	9
13	Software	Java	İleri 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

202

# GEÇİCİ VERİLER İLE ÇALIŞMAK

SQL Server'da sık kullanılan ve işlevselliği yüksek olan tarih işlemleri için farklı veri tipleri ve fonksiyonlar geliştirilmiştir. Tarih işlemleri temel anlamda basit gibi görünse de, uygulamaların doğru oluşturulabilmesi için önemli bir konudur.

Bu bölümde, genel olarak geliştiricilerin detaylarına hakim olmadığı ve yeni başlayanların detaylarını kavramakta zorlandığı tarih işlemlerini inceleyeceğiz.

Veri girişi sırasında kullanılan tarih ve saat özelliklerinin yanı sıra, çıktı olarak alınacak formatların belirlenmesi, işlenmesi, dönüştürülmesi gibi konuları detaylandıracıız.

## SQL SERVER TARİH / ZAMAN VERİ TIPLERİ

Tarih işlemleri SQL Server'da veri tipleri bazında desteklenmektedir. Yani, tarih işlemlerini gerçekleştirmek için, sütunlarda birçok farklı tarih veri tipi kullanılabilir. Bu veri tipleri temel anlamda benzer işlemleri gerçekleştirse de, özellik olarak farklılıklar vardır.

### DATE

Tarih tipinden veri saklamaya yarar. 01-01-0001 ile 31-12-9999 arasında tarih değeri alabilir.

**Format:**

YYYY-MM-DD

**Varsayılan Değer:**

1900-01-01

**Örnek :**

2007-05-08

Yukarıdaki formata uygun olarak bir değişken tanımı ve görüntülenmesini gerçekleştirelim.

```
DECLARE @date date = '2013-01-25';
SELECT @date AS '@date';
```

	@date
1	2013-01-25

**Date ile DateTime arasındaki fark:**

```
DECLARE @date date= '12-10-25';
DECLARE @datetime datetime= @date;
SELECT @date AS '@date', @datetime AS '@datetime';
```

	@date	@datetime
1	2025-12-10	2025-12-10 00:00:00.000

**TIME**

Saat tipinden veri saklamaya yarar. 5 baytlık depolama alanına sahiptir. Nano saniye bölümü 7 digit değerden oluşur. Tablo oluştururken nano saniye kısmında farklı değer belirterek nano saniye hassasiyeti belirlenebilmektedir.

**Time(3)** kullanımı, nano saniye değerinin 3 rakamlı hassasiyetini belirtir.  
**Time(7)** kullanımı ise, 7 rakamlı nano saniye hassasiyetini belirtir.

**Format :**

hh:mm:ss [.nnnnnnn]

**Varsayılan Değer :**

00:00:00

**Örnek :**

```
12:35:29.1234567
```

Örnek bir saat bilgisi üzerinde **Time** veri tipini kullanalım.

```
DECLARE @time time(7) = '12:34:54.1234567';
DECLARE @time1 time(1) = @time;
DECLARE @time2 time(2) = @time;
DECLARE @time3 time(3) = @time;
DECLARE @time4 time(4) = @time;
DECLARE @time5 time(5) = @time;
DECLARE @time6 time(6) = @time;
DECLARE @time7 time(7) = @time;

SELECT
    @time1 AS 'time(1)', @time2 AS 'time(2)', @time3 AS 'time(3)',
    @time4 AS 'time(4)', @time5 AS 'time(5)', @time6 AS 'time(6)',
    @time7 AS 'time(7)';
```

	time(1)	time(2)	time(3)	time(4)	time(5)	time(6)	time(7)
1	12:34:54.1	12:34:54.12	12:34:54.123	12:34:54.1235	12:34:54.12346	12:34:54.123457	12:34:54.1234567

Diğer veri tiplerine göre farklılıklarını gözlemleyelim.

**Time** ile **Datetime** arasındaki fark:

```
DECLARE @time time(4) = '12:15:04.1234';
DECLARE @datetime datetime= @time;
SELECT @time AS '@time', @datetime AS '@datetime';
```

	@time	@datetime
1	12:15:04.1234	1900-01-01 12:15:04.123

**SMALLDATETIME**

**Datetime** veri tipinin daha az kapsamlı halidir. Nano saniye değerini içermez ve 01-01-1900 ile 06-06-2079 tarihleri arasında değer alabilir. 4 baytlık veri tipidir.

**Format:**

```
YYYY-MM-DD hh:mm:ss
```

**Varsayılan Değer:**

1900-01-01 00:00:00

**Örnek Değer :**

2007-05-08 12:35:00

**SmallDateTime ile Date arasındaki fark:**

```
DECLARE @smalldatetime smalldatetime = '1955-12-13 12:43:10';
DECLARE @date date = @smalldatetime;
SELECT @smalldatetime AS '@smalldatetime', @date AS 'date';
```

	@smalldatetime	date
1	1955-12-13 12:43:00	1955-12-13

**DATETIME**

Tarih ve saat tipinden veri saklamaya yarar. 8 baytlık veri tipidir. 01-01-1753 ile 31-12-9999 arasında bir tarih değer alabilir. Aynı zamanda nanosaniye olarak 3 digit'i desteklemektedir.

**Format:**

YYYY-MM-DD hh:mm:ss[.nnn]

**Varsayılan Değer:**

1900-01-01 00:00:00

**Örnek :**

2013-05-05 12:35:29.123

**DateTime ile SmallDateTime arasındaki fark:**

```
DECLARE @smalldatetime smalldatetime = '1955-12-13 12:43:10';
DECLARE @datetime datetime = @smalldatetime;
SELECT @smalldatetime AS '@smalldatetime', @datetime AS '@datetime';
```

	@smalldatetime	@datetime
1	1955-12-13 12:43:00	1955-12-13 12:43:00.000

## DATETIME2

`DateTime` veritipinin aynısıdır. Bir farkı, nano saniye olarak 7 digit bilgisini içermesi, diğer ise 01-01-0001 ile 31-12-9999 tarihleri arasında değer almasıdır.

### Format:

YYYY-MM-DD hh:mm:ss [.nnnnnnn]

### Varsayılan Değer :

1900-01-01 00:00:00

### Örnek :

2007-05-08 12:35:29. 1234567

### DATETIME2 ile DATE arasındaki fark;

---

```
DECLARE @datetime2 datetime2(4) = '12-10-25 12:32:10.1234';
DECLARE @date date = @datetime2;
SELECT @datetime2 AS '@datetime2', @date AS 'date';
```

---

	@datetime2	date
1	2025-12-10 12:32:10.1234	2025-12-10

## DATETIMEOFFSET

**UTC** (*Coordinated Universal Time = Eşitlenmiş Evrensel Zaman*) bilgisi içeren tarih-saat veri tipidir.

### Format:

YYYY-MM-DD hh:mm:ss [.nnnnnnn] [+|-]hh:mm

### Varsayılan Değer:

1900-01-01 00:00:00 00:00

### Örnek :

2007-05-08 12:35:29.1234567 +12:15

**DateDateTimeOffset ile Date arasındaki fark:**

```
DECLARE @datetimeoffset datetimeoffset(4) = '12-10-25 12:32:10 +01:00';
DECLARE @date date= @datetimeoffset;
SELECT @datetimeoffset AS '@datetimeoffset ', @date AS 'date';
```

	@datetimeoffset	date
1	2025-12-10 12:32:10.0000 +01:00	2025-12-10

## GİRİ TARIH FORMATLARI

Veritabanı uygulamalarında gerçek veriler ile çalışırken, tarih-saat verilerinin yönetimi önem arz eder. Bir tarih işlemi için farklı istemcilerden gelebilecek farklı tarih-saat formatları veritabanında çeşitli sorgu ve prosedürler ile işlenerek veritabanının ilgili sütununa doğru ve tutarlı şekilde depolanması gereklidir.

Farklı istemcilere sahip ve çok dilli uygulamaların tek bir veritabanını kullanıyor olması tarih-saat verilerinin doğru yönetilmesini gerektirir. İngiltere'den İngilizce dili ile istemci kullanan ile Türkiye'den Türkçe istemci kullanan bir platformun ortak tarihler üzerinde işlemler yapabilmesi ve görüntülemesi için veritabanının doğru tasarılanması ve sorguların bu kriterlere göre optimize edilmesi gereklidir.

SQL Server, bu işlemleri kolaylaştırmak için birçok veri tipi ve fonksiyon kullanır. Ancak, öncelikle bir veritabanına ne tür tarih girişi yapılabileceğine bakalım.

Bu tür karmaşık yapıdaki mimarilerin doğru tarih-saat veri formatlarını kullanabilmesi için Uluslararası Standartlar Örgütü, ISO 8601 standartını oluşturmuştur.



ISO 8601 standartı, veri değişimi kapsamında; Gregorian takvimine göre tarihleri, 24 saatlik gösterime göre zamanı, zaman aralıklarını ve zaman aralıklarının yeniden gösterimi ile bu gösterimlerin formatlarını kapsar.

Standart tarih formatı olarak aşağıdaki format kullanılır.

yyyy-mm-dd hh:mi:ss.mmm

Başlangıçta anlamsız görünen bu dizilimin daha anlaşılabilir karşılığı aşağıdaki gibidir:

yıl-ay-gün saat:dakika:saniye:milisaniye

Şu anki tarihim aşağıdaki gibi gösterilebilir.

Ayrıştırma yapılmadan;

20130202 00:45:05

ISO 8601 standardına göre ise şu şekilde gösterilir.

2013-02-02 00:45:05

Sadece tarih:

20130202

2013-02-02

Sadece zaman:

00:45:05

SQL Server'da bu karmaşık görünen formatlama işlemlerini çözmek için, farklı dillere göre ayar değiştirme özelliği eklenmiştir.

Dışarıdan alınan bir tarih-saat değerini `CAST` ve `CONVERT` fonksiyonları kullanılarak, farklı birçok formata dönüştürmek mümkündür. Bu fonksiyonların kullanımını, ilerleyen bölümlerde tüm detaylarıyla işleyeceğiz.

Tarih formatında varsayılan kullanımın `yyyy-mm-dd` olduğunu belirtmişik. Ancak bu format da istege göre değiştirilebilir.

-- AY/GÜN/YIL

```
SET DATEFORMAT MDY           Sonuç : 2013-12-31 00:00:00.000
```

```
DECLARE @datevar datetime
```

```
SET @datevar = '12/31/13'
```

```
SELECT @datevar
```

```
GO
```

-- YIL/GÜN/AY

```
SET DATEFORMAT YDM           Sonuç : 2013-12-31 00:00:00.000
```

```
DECLARE @datevar datetime
```

```
SET @datevar = '13/31/12'
```

```
SELECT @datevar
```

```
GO
```

```
-- YIL/AY/GÜN
SET DATEFORMAT YMD          Sonuç : 2013-12-31 00:00:00.000
DECLARE @datevar datetime
SET @datevar = '13/12/31'
SELECT @datevar
GO

-- GÜN/AY/YIL
SET DATEFORMAT DMY          Sonuç : 2013-12-31 00:00:00.000
DECLARE @datevar datetime
SET @datevar = '31/12/13'
SELECT @datevar
```

---

Yukarıdaki tüm sorgularda `@datevar` değişkeninin değerine bakarsanız, hepsinde tarih değerlerinin farklı konumlandırıldığını görebilirsiniz. Ancak tüm sorgu çıktılarında aynı tarih formatı ile sonuç üretildi.

## SQL SERVER TARİH/SAAAT FONKSİYONLARI

Veritabanı işlemlerinde tarih ve saat gibi zaman bilgilerinin önemi yüksektir. Bu tür veriler farklı dil ve kültürlerde göre şekillenen ve formatlanması gereken verilerdir. Bu nedenle, SQL Server gibi büyük veritabanlarında tarih/saat işlemlerini gerçekleştirecek çok sayıda veri tipi ve fonksiyon vardır.

Bu bölümde, SQL Server'da tarih/saat işlemlerini gerçekleştiren fonksiyonları inceleyeceğiz.

### GETDATE

Sistemin anlık tarihini gösterir.

#### Söz Dizimi:

---

```
GETDATE()
```

---

#### Örnek:

---

```
SELECT GETDATE();
```

---

	(No column name)
1	2013-02-03 13:28:23.383

**GETDATE** fonksiyonu, genel olarak veritabanında en sık kullanılan tarih fonksiyonlarından biridir. Tablolarda **DEFAULT** olarak kullanılan **GETDATE** fonksiyonu, o anki sistem tarih-saat bilgisini otomatik olarak alarak veritabanına ekler.

## CAST VE CONVERT İLE TARİH FORMATLAMA

Tarih formatını ayarlamak için kullanılır.

### Söz Dizimi:

---

```
CONVERT(VARCHAR, Tarih, KOD)
```

---

### Örnek:

---

```
SELECT CONVERT(VARCHAR(16), GETDATE(), 100)
```

---

Tarih dönüştürme işlemlerinde **CONVERT** sıklıkla kullanıldığı gibi **CAST** fonksiyonu da kullanılır.

---

```
SELECT GETDATE() AS DonusturulmemisTarih,  
       CAST(GETDATE() AS NVARCHAR(30)) AS Cast_ile,  
       CONVERT(NVARCHAR(30), GETDATE(), 126) AS Convert_ile;
```

---

	DonusturulmemisTarih	Cast_ile	Convert_ile
1	2013-02-03 13:28:50.923	Feb 3 2013 1:28PM	2013-02-03T13:28:50.923

String olarak alınan bir değer, **CAST** ve **CONVERT** fonksiyonlarının her ikisiyle de DATETIME veri tipine dönüştürülebilir.

---

```
SELECT '2006-04-25T15:50:59.997' AS DonusturulmemisTarih,  
       CAST('2006-04-25T15:50:59.997' AS DATETIME) AS Cast_ile,  
       CONVERT(DATETIME, '2006-04-25T15:50:59.997', 126) AS Convert_ile;
```

---

	DonusturulmemisTarih	Cast_ile	Convert_ile
1	2006-04-25T15:50:59.997	2006-04-25 15:50:59.997	2006-04-25 15:50:59.997

Tarih işlemlerinde birçok farklı kültür ve tarih kullanım formatı olması nedeniyle, dönüştürme ve tarih verilerini işleme sırasında birçok farklı sorunla karşılaşılabilir.

Bu nedenle tarih biçimlerinin tamamını elde edebileceğiniz tüm yöntemleri göstermek adına, aşağıdaki tarih dönüştürme işlemlerini gerçekleştirelim.

### **CONVERT ile ilgili farklı formatlama kodları:**

---

```

SELECT CONVERT(VARCHAR, GETDATE(), 0) Sonuç : Feb 1 2013 10:06AM
SELECT CONVERT(VARCHAR, GETDATE(), 1) Sonuç : 02/01/13
SELECT CONVERT(VARCHAR, GETDATE(), 2) Sonuç : 13.02.01
SELECT CONVERT(VARCHAR, GETDATE(), 3) Sonuç : 01/02/13
SELECT CONVERT(VARCHAR, GETDATE(), 4) Sonuç : 01.02.13
SELECT CONVERT(VARCHAR, GETDATE(), 5) Sonuç : 01-02-13
SELECT CONVERT(VARCHAR, GETDATE(), 6) Sonuç : 01 Feb 13
SELECT CONVERT(VARCHAR, GETDATE(), 7) Sonuç : Feb 01, 13
SELECT CONVERT(VARCHAR, GETDATE(), 8) Sonuç : 10:08:01
SELECT CONVERT(VARCHAR, GETDATE(), 9) Sonuç : Feb 1 2013 10:08:10:327AM
SELECT CONVERT(VARCHAR, GETDATE(), 10) Sonuç : 02-01-13
SELECT CONVERT(VARCHAR, GETDATE(), 11) Sonuç : 13/02/01
SELECT CONVERT(VARCHAR, GETDATE(), 12) Sonuç : 130201
SELECT CONVERT(VARCHAR, GETDATE(), 13) Sonuç : 01 Feb 2013 10:08:45:857
SELECT CONVERT(VARCHAR, GETDATE(), 14) Sonuç : 10:08:54:127
SELECT CONVERT(VARCHAR, GETDATE(), 20) Sonuç : 2013-02-01 10:09:07
SELECT CONVERT(VARCHAR, GETDATE(), 21) Sonuç : 2013-02-01 10:09:23.973
SELECT CONVERT(VARCHAR, GETDATE(), 22) Sonuç : 02/01/13 10:09:32 AM
SELECT CONVERT(VARCHAR, GETDATE(), 23) Sonuç : 2013-02-01
SELECT CONVERT(VARCHAR, GETDATE(), 24) Sonuç : 10:09:49
SELECT CONVERT(VARCHAR, GETDATE(), 25) Sonuç : 2013-02-01 10:09:57.257
SELECT CONVERT(VARCHAR, GETDATE(), 100) Sonuç : Feb 1 2013 10:10AM
SELECT CONVERT(VARCHAR, GETDATE(), 101) Sonuç : 02/01/2013
SELECT CONVERT(VARCHAR, GETDATE(), 102) Sonuç : 2013.02.01
SELECT CONVERT(VARCHAR, GETDATE(), 103) Sonuç : 01/02/2013
SELECT CONVERT(VARCHAR, GETDATE(), 104) Sonuç : 01.02.2013
SELECT CONVERT(VARCHAR, GETDATE(), 105) Sonuç : 01-02-2013
SELECT CONVERT(VARCHAR, GETDATE(), 106) Sonuç : 01 Feb 2013
SELECT CONVERT(VARCHAR, GETDATE(), 107) Sonuç : Feb 01, 2013
SELECT CONVERT(VARCHAR, GETDATE(), 108) Sonuç : 10:12:24
SELECT CONVERT(VARCHAR, GETDATE(), 109) Sonuç : Feb 1 2013 10:12:31:237AM
SELECT CONVERT(VARCHAR, GETDATE(), 110) Sonuç : 02-01-2013
SELECT CONVERT(VARCHAR, GETDATE(), 111) Sonuç : 2013/02/01
SELECT CONVERT(VARCHAR, GETDATE(), 112) Sonuç : 20130201
SELECT CONVERT(VARCHAR, GETDATE(), 113) Sonuç : 01 Feb 2013 10:13:01:103

```

```
SELECT CONVERT(VARCHAR, GETDATE(), 114) Sonuç : 10:13:14:437
SELECT CONVERT(VARCHAR, GETDATE(), 120) Sonuç : 2013-02-01 10:13:21
SELECT CONVERT(VARCHAR, GETDATE(), 121) Sonuç : 2013-02-01 10:13:27.237
SELECT CONVERT(VARCHAR, GETDATE(), 126) Sonuç : 2013-02-01T10:13:34.317
SELECT CONVERT(VARCHAR, GETDATE(), 127) Sonuç : 2013-02-01T10:13:41.897
```

---

Farklı ihtiyaçlara göre şekillenecek tarih formatı gereksinimi, yukarıdaki formatlar ile karşılanabilir.

## FORMAT

Verilen tarihi istenen formata dönüştürerek string olarak verir.

### Söz Dizimi:

---

```
FORMAT(tarih_zaman, format)
```

---

### Örnek:

---

```
SELECT FORMAT(GETDATE(), 'yyyy.MM.d HH:MM:ss');
```

---

**FORMAT** fonksiyonu, genel olarak tarih bilgilerini farklı kültür formatlarında görüntülemek için kullanılır.

	(No column name)
1	2013.02.3 13:02:27

Belirli bir tarih bilgisini farklı kültür bilgilerine göre görüntüleyelim.

---

```
DECLARE @tarih DATETIME = GETDATE()
SELECT FORMAT ( @tarih, 'd', 'tr-TR' ) AS 'Türkçe'
      ,FORMAT ( @tarih, 'd', 'en-US' ) AS 'Amerikan İngilizcesi'
      ,FORMAT ( @tarih, 'd', 'en-gb' ) AS 'İngiltere İngilizcesi'
      ,FORMAT ( @tarih, 'd', 'de-de' ) AS 'Almanca'
      ,FORMAT ( @tarih, 'd', 'zh-cn' ) AS 'Çince';
```

---

	Türkçe	Amerikan İngilizcesi	İngiltere İngilizcesi	Almanca	Çince
1	03.02.2013	2/3/2013	03/02/2013	03.02.2013	2013/2/3

Küçük harfli `d` parametresini büyük harf `D` ile değiştirdiğimizde ise; ay ve gün bilgilerini ilgili dillere çevirerek açıkça görüntüleyecektir.

---

```
DECLARE @tarih DATETIME = GETDATE()
SELECT FORMAT ( @tarih, 'D', 'tr-TR' ) AS 'Türkçe'
      ,FORMAT ( @tarih, 'D', 'en-US' ) AS 'Amerikan İngilizcesi'
      ,FORMAT ( @tarih, 'D', 'en-gb' ) AS 'İngiltere İngilizcesi'
      ,FORMAT ( @tarih, 'D', 'de-de' ) AS 'Almanca'
      ,FORMAT ( @tarih, 'D', 'zh-cn' ) AS 'Çince';
```

---

	Türkçe	Amerikan İngilizcesi	İngiltere İngilizcesi	Almanca	Çince
1	03 Şubat 2013 Pazar	Sunday, February 03, 2013	03 February 2013	Sonntag, 3. Februar 2013	2013年2月3日

Farklı tarih işlemleri için `FORMAT` fonksiyonu aşağıdaki formatlarda kullanılabilir.

---

```
SELECT FORMAT ( GETDATE(), 'd', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'dd', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'ddd', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'dddd', 'tr-TR' ) AS Turkiye;
```

---

Turkiye	
1	03.02.2013
<hr/>	
1	03
<hr/>	
1	Paz
<hr/>	
1	Pazar

Turkiye	
1	03 Şubat
<hr/>	
1	Turkiye
<hr/>	
1	33
<hr/>	
1	Turkiye
<hr/>	
1	33
<hr/>	
1	Turkiye
<hr/>	
1	33

---

```
SELECT FORMAT ( GETDATE(), 'm', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'mm', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'mmm', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'mmmm', 'tr-TR' ) AS Turkiye;
```

---

---

```
SELECT FORMAT ( GETDATE(), 'Y', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'YY', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( GETDATE(), 'YYY', 'tr-TR' ) AS Turkiye;
```

---

Turkiye	
1	Şubat 2013
Turkiye	
1	13
Turkiye	
1	2013

## FORMAT FONKSİYONUNUN FARKLI ÜLKE PARA BİRİMLERİ İLE KULLANIMI

SQL Server'da maaş, ücret, ödeme gibi işlemler için saklanan para birimlerini farklı ülkelerin farklı para birimlerine göre dönüştürme ihtiyacı duyulur. Bu tür durumlarda SQL Server varsayılan destek sağlar.

---

```
DECLARE @para INT = 500;
SELECT FORMAT ( @para, 'c', 'tr-TR' ) AS Turkiye;
SELECT FORMAT ( @para, 'c', 'en-US' ) AS ABD;
SELECT FORMAT ( @para, 'c', 'fr-FR' ) AS Fransa;
SELECT FORMAT ( @para, 'c', 'de-DE' ) AS Almanca;
SELECT FORMAT ( @para, 'c', 'zh-cn' ) AS Cin;
```

---

Para değerinin sonuna 0 (sıfır) konulabilir. Bu durumda, sorgulamanın ikinci parametresi olan 'c' parametresi şu şekilde kullanılacaktır.

---

```
DECLARE @para INT = 50
SELECT FORMAT(@para,'c') AS Para;
SELECT FORMAT(@para,'c1') AS Para;
SELECT FORMAT(@para,'c2') AS Para;
SELECT FORMAT(@para,'c3') AS Para;
```

---

Turkiye	
1	500,00 ₺
ABD	
1	\$500,00
Fransa	
1	500,00 €
Almanca	
1	500,00 €
Cin	
1	¥500,00

Para	
1	\$50,00
Para	
1	\$50,00
Para	
1	\$50,00
Para	
1	\$50,00

Bilindiği gibi Türk Lirası için belirlenen para simgesi, 1 Mart 2012 tarihinde resmi olarak kullanıma sunulduğu duyuruldu. Bu resmi duyurudan sonra global ve yerel yazılımlar, Türk Lirası simgesini kullanmak için entegrasyon yazılımları hazırladı. Microsoft'ta TL simgesini bir güncelleştirme ile duyurarak SQL Server ve Windows'ta kullanılabilir hale getirdi.

Windows'ta, dolayısıyla SQL Server'da, TL simgesini kullanabilmek için (**Alt+Gr+T**) tuş kombinasyonlarını kullanmanız gereklidir.

Aşağıdaki Windows güncelleştirme bağlantısından ilgili güncelleştirmeyi bilgisayarınıza kurarak, TL simgesinin Windows ve SQL Server tarafından desteklenmesini sağlamanız gereklidir.

TL Simgesi: 

İndirme Bağlantısı: <http://support.microsoft.com/kb/2739286>

**FORMAT** fonksiyonu farklı bazı işlemler için de kullanılabilir. Bu işlemleri bir kaç örnek vererek açıklayalım.

Yüzde hesaplamaları, bilimsel ve heksadesimal (*hexa*) gibi işlemler için kullanılabilir.

---

```
DECLARE @var INT = 50
SELECT FORMAT(@var,'p') AS Yüzde;           --(P = Percentage)
SELECT FORMAT(@var,'e') AS Bilimsel;         --(E = Scientific)
SELECT FORMAT(@var,'x') AS Hexa;
SELECT FORMAT(@var,'x4') AS Hexal;
```

---

O anki sistem tarihini, kendi belirlediğiniz tarih formatı ve kültür formatlarında görüntüleyebilir.

---

```
SELECT FORMAT (GETDATE(), N'dddd MMMM dd, yyyy', 'tr-TR') AS Turkce;
SELECT FORMAT (GETDATE(), N'dddd MMMM dd, yyyy', 'en-US') AS Ingilizce;
SELECT FORMAT (GETDATE(), N'dddd MMMM dd, yyyy', 'hi') AS HindistanDili;
SELECT FORMAT (GETDATE(), N'dddd MMMM dd, yyyy', 'gu') AS GujaratDili;
SELECT FORMAT (GETDATE(), N'dddd MMMM dd, yyyy', 'zh-cn') AS Cince;
```

---

	Yüzde
1	5,000.00 %
<hr/>	
	Bilimsel
1	5.000000e+001
<hr/>	
	Hexa
1	32
<hr/>	
	Hexa1
1	0032

**FORMAT** fonksiyonunun, veritabanındaki veriler üzerinde kullanımı da basittir.

---

```
SELECT ModifiedDate, ListPrice,
       FORMAT(ModifiedDate, N'ddd MMMM dd, yyyy','tr') TR_DegistirmeTarih,
       FORMAT(ListPrice, 'c','tr') TR_UrunFiyat
FROM Production.Product;
```

---

	ModifiedDate	ListPrice	TR_DegistirmeTarih	TR_UrunFiyat
1	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
2	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
3	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
4	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
5	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
6	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺
7	2008-03-11 10:01:36.827	0,00	Salı Mart 11, 2008	0,00 ₺

## DATEPART

Tarih ile ilgili bazı rakamsal bilgiler alınmasını sağlar. Verilen tarihin gün, ay, yıl gibi parçalar halinde geri almak için kullanılır.

### Söz Dizimi:

---

DATEPART (parca\_datepart, tarih)

---

### Örnek:

---

```
SELECT DATEPART(WEEKDAY, GETDATE());
```

---

	(No column name)
1	1

**DATEPART** fonksiyonu ile kullanılan parçalar aşağıdaki gibidir:

### **YY, YYYY YA DA YEAR**

Verilen tarih içerisindeki yıl bilgisini almayı sağlar.

---

SELECT DATEPART(YY, GETDATE());	Sonuç : 2013
SELECT DATEPART(YYYY, GETDATE());	Sonuç : 2013
SELECT DATEPART(YEAR, GETDATE());	Sonuç : 2013

---

### **QQ, Q YA DA QUARTER**

Verilen tarihin çeyreğini almak için kullanılır. 12 ayın 4'e bölümü sonucunda ortaya çıkan her 3 ay bir çeyrektir. İlk 3 ay ilk çeyrek, son 3 ay son çeyrektir.

---

SELECT DATEPART(QQ, GETDATE());	Sonuç : 1
SELECT DATEPART(Q, GETDATE());	Sonuç : 1
SELECT DATEPART(QUARTER, GETDATE());	Sonuç : 1

---



Sorgu çalıştırıldığında ilk çeyrekte olduğumuz için bu sonuç dönmektedir.

### **MM, M YA DA MONTH**

Verilen tarihin ay bilgisini almak için kullanılır.

---

SELECT DATEPART(MM, GETDATE());
SELECT DATEPART(M, GETDATE());
SELECT DATEPART(MONTH, GETDATE());

---

### **DY, Y YA DA DAYOFYEAR**

Yılın kaçinci günü olduğunu almak için kullanılır.

---

SELECT DATEPART(DY, GETDATE());
SELECT DATEPART(Y, GETDATE());
SELECT DATEPART(DAYOFYEAR, GETDATE());

---

**DD, D YA DA DAY**

Verilen tarihin gün bilgisini almak için kullanılır.

---

```
SELECT DATEPART(DD, GETDATE());
SELECT DATEPART(D, GETDATE());
SELECT DATEPART(DAY, GETDATE());
```

---

**WK YA DA WW**

Verilen tarihin hafta bilgisini almak için kullanılır.

---

```
SELECT DATEPART(WK, GETDATE());
SELECT DATEPART(WW, GETDATE());
SELECT DATEPART(WEEK, GETDATE());
```

---

**DW YA DA WEEKDAY**

Haftanın kaçinci günü olduğunu almak için kullanılır.

---

```
SELECT DATEPART(DW, GETDATE());
SELECT DATEPART(WEEKDAY, GETDATE());
```

---

**HH YA DA HOUR**

Verilen tarih-zamanın saat bilgisini almak için kullanılır.

---

```
SELECT DATEPART(HH, GETDATE());
SELECT DATEPART(HOUR, GETDATE());
```

---

**MI, N YA DA MINUTE**

Verilen tarih-zamanın dakika bilgisini almak için kullanılır.

---

```
SELECT DATEPART(MI, GETDATE());
SELECT DATEPART(N, GETDATE());
SELECT DATEPART(MINUTE, GETDATE());
```

---

**SS, S YA DA SECOND**

Verilen tarih-zamanın saniye bilgisini almak için kullanılır.

---

```
SELECT DATEPART(SS, GETDATE());
SELECT DATEPART(S, GETDATE());
SELECT DATEPART(SECOND, GETDATE());
```

---

**MS, MCS MILLISECOND**

Verilen tarih-zamanın milisaniye bilgisini almak için kullanılır.

---

```
SELECT DATEPART(MS, GETDATE());
SELECT DATEPART(MILLISECOND, GETDATE());
```

---

**ISDATE**

Tarih formatının geçerliliğini denetler. Bu fonksiyonun kullanımında kültür kodu önemlidir. Tarih formatlama işlemlerinde bir kültür kodu (tr-TR, en-EN vb.) ile başarılı şekilde çalışan tarih formatı bir başkasında çalışmamayabilir. Veri işlemlerinde hataya sebep olmamak için bazen tarih formatlarını test etmek ve geçerliliğini onayladıkтан sonra işleme almak gerekebilir. Bu tür durumlarda **ISDATE** fonksiyonu kullanılabilir.

Tarih formatı geçerliyse 1, geçerli değilse 0 değerini döndürür.

**Söz Dizimi:**


---

```
ISDATE(tarih)
```

---

Varsayılan olarak SQL Server dil ayarı **us\_english**, yani Amerikan İngilizcesi ile kullanılır. Bu ayarda iken aşağıdaki sorguları çalıştıralım.

---

```
SELECT ISDATE('02/15/2013');          Sonuç : 1
SELECT ISDATE('15/02/2013');          Sonuç : 0
```

---

İlk sorgudaki format, 1 değerini döndürdü. Yani, ilk tarih formatımız geçerlidir. Ancak ikinci sorgudaki format, 0 değerini döndürerek geçerli olmadığını bildirdi.

Şimdi ise dil kodunu değiştirelim.

---

```
SET LANGUAGE Turkish
```

---

Dil ayarı 'Türkçe' olarak değiştirildi.

Yukarıda geçerlilik testi yaptığımız iki tarihi de aynı şekilde tekrar sorgulayalım.

---

```
SELECT ISDATE('02/15/2013');      Sonuç : 0
SELECT ISDATE('15/02/2013');      Sonuç : 1
```

---

Sorgular aynı, ancak sonuçlar farklı. Bunun nedeni, farklı dillerde farklı formatların kullanılıyor olmasıdır.

Yazılımsal olarak geçerlilik testi şu şekilde yapılabilir.

---

```
IF ISDATE('15/02/2013') = 1
    PRINT 'GEÇERLİ'
ELSE
    PRINT 'GEÇERSİZ'
```

---

**GEÇERLİ**

Şu anki geçerli dil Türkçe olduğu için bu format geçerli olarak sonuç döndürdü.

Dil ayarını değiştirelim.

---

```
SET LANGUAGE us_english
```

---

Changed language setting to us\_english.

Aynı soruyu tekrar çalışıralım.

---

```
IF ISDATE('15/02/2013') = 1
    PRINT 'GEÇERLİ'
ELSE
    PRINT 'GEÇERSİZ'
```

---

**GEÇERSİZ**

Benzer şekilde farklı diller ile şu şekilde farklı sonuçlar üretecek bir örnek oluşturalım.

---

```
SET LANGUAGE Turkish;
SELECT ISDATE('15/02/2013'); -- Sonuç : 1
SET LANGUAGE English;
SELECT ISDATE('15/02/2013'); -- Sonuç : 0
SET LANGUAGE Hungarian;
SELECT ISDATE('15/2013/02'); -- Sonuç : 0
SET LANGUAGE Swedish;
SELECT ISDATE('2013/15/02'); -- Sonuç : 0
SET LANGUAGE Italian;
SELECT ISDATE('15/02/2013'); -- Sonuç : 1
```

---

Dil ayarını değiştirdikten sonra haliyle sonuçta değişti. Bu örnek, tarih işlemlerinin aslında göründüğü kadar kolay olmadığını ve gelişmiş veritabanlarında ne kadar önemli ve kritik olabileceğini gösterir.

Çok dilli veritabanı mimarilerinde bir çok para birimi ve dil ile uğraşmak cidden çok can sıkıcı hal alabilir. Bu nedenle tarih işlemlerine hakim olmanız işleri kolaylaştıracaktır.

## **DATEADD**

Verilen tarihin ay, gün ya da yıl değerini artırmak ya da azaltmak için kullanılır.

### **Söz Dizimi:**

---

```
DATEADD ( datepart , sayı, tarih )
```

---

- **Datepart:** Değişiklik yapılacak kısım.
- **Number:** Eklenecek ya da çıkarılacak sayı.
- **Date:** Değişiklik yapılacak tarih.

### **Örnek:**

---

01.01.2013 tarihine 2 yıl ekleyelim.

---

```
SELECT DATEADD(YY, 2, '01.01.2013');
```

---

Belirtilen tarihin 2 ay ekleyelim.

---

```
SELECT DATEADD(MM, 2, '01.01.2013');
```

---

Belirtilen tarihin çeyreğini değiştirelim.

---

```
SELECT DATEADD(QQ, 3, '01.01.2013');
```

---

Belirtilen tarihin ayını değiştirelim. 3 ay öncesine gidelim.

---

```
SELECT DATEADD(MM, -3, '01.01.2013');
```

---

**DATENAME** fonksiyonunda kullanılan tüm parçalar **DATEADD** fonksiyonunda da kullanılarak azaltma ya da artırma işlemleri yapılabilir.

Bu parçaları özetle hatırlatmak gereklidir;

- **dy / y:** Yılın gününü değiştirmek için kullanılır.
- **dd / d:** Günü değiştirmek için kullanılır.
- **wk / ww:** Haftayı değiştirmek için kullanılır.
- **dw:** Haftanın gününü değiştirmek için kullanılır.
- **hh:** Saati değiştirmek için kullanılır.
- **mi / n:** Dakikayı değiştirmek için kullanılır.
- **ss / s:** Saniyeyi değiştirmek için kullanılır.
- **ms:** Milisaniyeyi değiştirmek için kullanılır.

## **DATEDIFF**

Verilen iki tarih arasındaki **Datepart** parametresi farkını verir.

### **Söz Dizimi:**

---

```
DATEDIFF(datepart, baslangic_tarih, bitis_tarih)
```

---

### **Örnek:**

İki tarih arasındaki yıl farkını bulalım.

---

```
SELECT DATEDIFF(YY,'01.01.2012','01.01.2013');
```

---

İki tarih arasındaki ay farkını bulalım.

---

```
SELECT DATEDIFF(MONTH,'01.01.2012','01.01.2013');
```

---

İki tarih arasındaki hafta farkını bulalım.

---

```
SELECT DATEDIFF(WK,'01.01.2012','01.01.2013');
```

---

İki tarih arasındaki gün farkını bulalım.

---

```
SELECT DATEDIFF(DD,'01.01.2012','01.01.2013');
```

---

Bu ve benzer işlemler için diğer fonksiyonlarda kullanılan tüm tarih datepart'ları (*parçaları*) kullanılabilir.

Örneğin; bu tarihler arası, yani bir yılın kaç saniye olduğunu hesaplayalım.

---

```
SELECT DATEDIFF(ss,'01.01.2012','01.01.2013');
```

---

## DATENAME

**Datepart** olarak verilen parametrenin adını döndürür.

**Söz Dizimi:**

---

DATENAME(datepart, tarih)

---

**Örnek:**

Bulduğumuz tarihin yıl bilgisini alalım.

---

```
SELECT DATENAME(YY, GETDATE());
```

---

Bulduğumuz tarihin ay bilgisini alalım.

---

```
SELECT DATENAME(YY, GETDATE());
```

---

Benzer şekilde diğer tüm **Datepart** parametreleri kullanılarak ilgili bilgiler alınabilir.

Bulunduğumuz zamanın milisaniye bilgisini alalım.

---

```
SELECT DATENAME(MS, GETDATE());
```

---

## DAY

Verilen bir tarihin sadece gün değerini alır.

### Söz Dizimi:

---

```
DAY(GETDATE())
```

---

### Örnek:

---

```
SELECT DAY(GETDATE());
```

---

## MONTH

Verilen bir tarihin sadece ay değerini alır.

### Söz Dizimi:

---

```
MONTH(GETDATE())
```

---

### Örnek:

---

```
SELECT MONTH(GETDATE());
```

---

## YEAR

Verilen bir tarihin sadece yıl değerini alır.

### Söz Dizimi:

---

```
YEAR(GETDATE())
```

---

### Örnek:

---

```
SELECT YEAR(GETDATE());
```

---

## **DATEFROMPARTS**

Tarihin el ile verilmesini sağlar ve tarih döndürür.

### **Söz Dizimi:**

---

DATEFROMPARTS(yıl, ay, gün)

---

### **Örnek:**

---

SELECT DATEFROMPARTS(2013, 1, 1);

---

## **DATETIMEFROMPARTS**

Tarih zamanın el ile verilmesini sağlar.

### **Söz Dizimi:**

---

DATETIMEFROMPARTS(yıl, ay, gün, saat, dakika, saniye, salise, milisaniye)

---

### **Örnek:**

---

SELECT DATETIMEFROMPARTS(2013, 02, 01, 17, 37, 12, 997);

---

Bu tür el ile işlem yapan fonksiyonlar genel olarak uygulamalardan ziyade, veri ve soru testlerinde kullanılabilir.

## **SMALLDATETIMEFROMPARTS**

`SmallDateTime`'ın el ile verilmesini sağlar ve tarih döndürür.

### **Söz Dizimi:**

---

SMALLDATETIMEFROMPARTS(yıl, ay, gün, saat, dakika)

---

### **Örnek:**

---

SELECT SMALLDATETIMEFROMPARTS(2013, 2, 1, 17, 45)

---

## TIMEFROMPARTS

El ile verilen bir zamanın, zaman tipinde bir değer olarak döndürülmesini sağlar.

### Söz Dizimi:

---

TIMEFROMPARTS (saat, dakika, saniye, milisaniye, ondalık)

---

### Örnek:

---

SELECT TIMEFROMPARTS (17, 25, 55, 5, 1);

---

**TIMEFROMPARTS**'ın aldığı son parametre, bir önceki parametrenin ondalık değerini belirtmek için kullanılır. Dördüncü parametre tek basamaklı ise beşinci parametre 1 değerini alır. Dördüncü parametre iki basamaklı ise beşinci parametre iki değerini alır. Son parametre on basamağa kadar değer alabilir.

---

SELECT TIMEFROMPARTS (17, 25, 55, 5, 1);

SELECT TIMEFROMPARTS (17, 25, 55, 50, 2);

SELECT TIMEFROMPARTS (17, 25, 55, 500, 3);

---

Yukarıdaki kullanım doğru olmakla birlikte, aşağıdaki kullanımı hata üretecektir.

---

SELECT TIMEFROMPARTS (17, 25, 55, 5000, 3);

---

**TIMEFROMPARTS** fonksiyonu, milisaniye değerinin maksimum 7 digit olması nedeniyle, 7 ondalık basamağına kadar değer alabilir.

---

SELECT TIMEFROMPARTS (17, 25, 55, 500000, 7);

---

Son parametre 0 (*sıfır*) ya da **NULL** değer de içermemz.

## EOMONTH

Verilen bir tarihin ayının kaç çekiğini verir

### Söz Dizimi:

---

EOMONT(tarih)

EOMONT(tarih, kod)

---

**Örnek:**


---

```
SELECT EOMONTH (GETDATE ()) ;
```

---

Örnek olarak belirtilen `EOMONTH (GETDATE ())` kullanımı ile sistem tarihindeki ay bilgisinin son gününün tam tarihini verecektir.

---

```
SELECT EOMONTH (GETDATE ()) ;
```

---

`EOMONTH` fonksiyonu ile şu anki tarihin kaç gün çektiği öğrenilebileceği gibi, önceki, sonraki ya da belirtilen bir başka ayın da kaç gün çektiği öğrenilebilir.

---

```
DECLARE @date DATETIME = GETDATE ();
SELECT EOMONTH (@date, -1) AS 'Önceki Ay';
SELECT EOMONTH (@date) AS 'Şimdiki Ay';
SELECT EOMONTH (@date, 1) AS 'Sonraki Ay';
```

---

Önceki Ay	
1	2013-01-31
Şimdiki Ay	
1	2013-02-28
Sonraki Ay	
1	2013-03-31

Yukarıdaki örnek ile ilk olarak bir önceki ayın son gün bilgisi, sonra şuan bulunan ay ve son olarak bir sonraki ayın son gün bilgisi listelenir.

**SYSDATETIME**

`DateTIme2` veri tipinde, sistemde anlık tarih ve zaman bilgisini döndürür. `GETDATE` fonksiyonu 3 digit milisaniye değeri döndürürken, `SYSDATETIME` fonksiyonu 7 digit'in tamamını döndürür.

**Söz Dizimi:**


---

```
SELECT SYSDATETIME
```

---

**Örnek:**


---

```
SELECT GETDATE () [GetDate], SYSDATETIME () [SysDateTime]
```

---

	GetDate	SysDateTime
1	2013-02-03 13:45:33.813	2013-02-03 13:45:33.8157828

## SYSUTCDATETIME

UTC zamanını `DateTime2` veri tipinde döndürür.

### Söz Dizimi:

---

`SYSUTCDATETIME()`

---

### Örnek:

---

`SELECT GETDATE(), SYSUTCDATETIME();`

---

	(No column name)	(No column name)
1	2013-02-03 13:46:13.217	2013-02-03 11:46:13.2188881

## SYSDATETIMEOFFSET

UTC'ye göre offset bilgisi ile yerel SQL Server saatini döndürür.

### Söz Dizimi:

---

`SYSDATETIMEOFFSET()`

---

### Örnek:

---

`SELECT SYSDATETIMEOFFSET() OffSet;`  
`SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '-04:00') 'OffSet -4';`  
`SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '-02:00') 'OffSet -2';`  
`SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '+00:00') 'OffSet +0';`  
`SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '+02:00') 'OffSet +2';`  
`SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '+04:00') 'OffSet +4';`

---

Offset
2013-02-03 13:46:58.3020805 +02:00
OffSet -4
2013-02-03 13:46:58.3020805 -04:00
OffSet -2
2013-02-03 13:46:58.3020805 -02:00
OffSet +0
2013-02-03 13:46:58.3020805 +00:00
OffSet +2
2013-02-03 13:46:58.3020805 +02:00
OffSet +4
2013-02-03 13:46:58.3020805 +04:00

## **SWITCHOFFSET**

Herhangi bir tarihi, başka bölge tarihine dönüştürmek için kullanılır. Girdi tarihte OffSet bilgisinin bulunması gereklidir.

### **Söz Dizimi:**

---

```
SWITCHOFFSET (DATETIMEOFFSET, zaman_dilimi)
```

---

### **Örnek:**

**OffsetTest** isimli tablo oluşturalım.

---

```
CREATE TABLE dbo.OffsetTest (
    ColDatetimeoffset datetimeoffset
)
;
```

---

Bir tarih verisi ekleyelim.

---

```
INSERT INTO dbo.OffsetTest VALUES ('2013-02-01 8:25:50.71345 -5:00');
```

---

**SWITCHOFFSET** fonksiyonunu test edelim.

---

```
SELECT SWITCHOFFSET (ColDatetimeoffset, '-08:00') FROM dbo.OffsetTest;
SELECT ColDatetimeoffset FROM dbo.OffsetTest;
```

---

(No column name)	
1	2013-02-01 05:25:50.7134500 -08:00
ColDatetimeoffset	
1	2013-02-01 08:25:50.7134500 -05:00

## **TODATETIMEOFFSET**

Herhangi bir tarihi koruyarak sadece UTC OffSet değerini ekler ya da varsa değiştirir.

### **Söz Dizimi:**

---

```
TODATETIMEOFFSET()
```

---

**Örnek:**

---

```
SELECT TODATETIMEOFFSET(SYSDATETIMEOFFSET(), '-05:00') AS TODATETIME;
```

---

	TODATETIME
1	2013-02-03 13:50:25.2232926 -05:00

**GETUTCDATE**

Anlık olarak Greenwich, yani GMT tarih-saat bilgisini döndürmeye yarar.

**Söz Dizimi:**

---

```
GETUTCDATE()
```

---

**Örnek:**

---

```
SELECT GETUTCDATE()
```

---

	(No column name)
1	2013-02-03 11:50:54.213



# VIEW'LERLE ÇALIŞMAK

7

SQL Server içerisinde hazırlanan sorgular basit sorgular olabildiği gibi birden fazla tablodan veri çeken, karmaşık sorgulardan da olabilmektedir. View'ler veritabanındaki veriler için hazırlanan `SELECT` sorgularına sanal bir görünüm oluşturmak için kullanılır. Yani tablodaki gerçek verileri içermezler. `SELECT` ifadelerinin uzun ve `JOIN` gibi karmaşık sorguları içерdiği durumlarda, bu sorguların kolay anlaşılabilir, kullanılabilir ve yönetilebilir olması için view'ler kullanılır. Aynı zamanda uygulamalarımızda kullandığımız SQL sorgularının içeriğinin son kullanıcı tarafından görüntülenmesini engellemek için de kullanılabilir. View içeriğini şifreleyerek hem SSMS hem de SQL tarafından, bu view'in içeriğinin görüntülenmesini engelleyebiliriz. View'ler sanılanın aksine, sadece `SELECT` işlemlerinin gerçekleştirildiği bir yapı değildir. Oluşturulan bir view üzerinde, belli kurallar ve kısıtlamalara uymak şartıyla DML sorguları (`Insert`, `Update`, `Delete`) gerçekleştirilebilir. Bu özelliklerin tamamını bölümün ilerleyen konularında detaylarıyla inceleyeceğiz.

## VIEW'LER NEDEN KULLANILIR?

SQL Server'da view kullanımı doğru yapılmalıdır. Her işlemde view kullanmak doğru değildir ve performansı etkileyebilir.

View'lerin kullanılması gereken durumlardan bazıları şunlardır;

- Uzun veritabanı sorgularını tekrarlamadan önde geçmek.
- Tabloda son kullanıcının erişmemesi gereken sütunları son kullanıcından gizleyip sadece gerekli sütunları göstermek.

- Sorgulama hızını artırmak.
- Parçalanmış tablolar ve Linked Server kavramlarını anlamak ve avantajlarından yararlanmak.
- Sorguları şifreleyerek son kullanıcının sorgu detaylarını görmesini engellemek.

## **VİEW TÜRLERİ**

- Regular View

Sadece tanımlamayı gösteren, veritabanında gerçek verinin saklanmadığı view'ler.

- Indexed View

Tanımlanan indeks ile, view'den donecek kayıtlar veritabanında saklanır. Sorgu performansı yüksektir.

- Distributed Partitioned View

Veritabanı işlem yükünü sunucuya dağıtmaya yarar. SQL Server verilerini bölgümler (*partitioning*). Yüksek veri içeren, büyük veritabanlarında performansı artırır.

## **ALTERNATİFLER**

View'ler genel olarak `SELECT` sorgularını kısaltmak ya da farklı formatlarda gösterebilme için kullanılabilir. Ancak programsal tarafta farklı özelliklere ihtiyacımız olabilir. Bu durumda view'in yaptığı işi gerçekleştirecek ancak view ile yapılamayan işlemleri de bilmemizde fayda var.

Tecrübeli bir geliştirici takım çantasındaki araçların olumlu ve olumsuz yönlerini çok iyi bilir.

- View'leri sadece takma isimler vermek için kullanıyorsanız sizin ihtiyacınızı Synonyms nesnesi görebilir. Synonyms nesnesi, herhangi bir veritabanı nesnesi için lakap olarak kullanılabilir.
- View kullanma amacınız sorguları kısa hale getirerek kullanmak olabilir. Ya da view içinde oluşturduğunuz `SELECT` sorgusundaki `WHERE` filtresinin, dinamik olarak dışarıdan göndereceğiniz değere göre sorgu sonucu getirmesini isteyebilirsiniz.

Bu ve bunun gibi sorgularınızda T-SQL blokları (**if-else** ya da döngüler vb.) kullanmak için kullanmanız gereken nesne Stored Procedure'lerdir. Bunlara kısaca **Sproc** denir. Önceden derlenmiş sproc'lar SQL Server'ın başlatılmasıyla birlikte kullanıma hazır hale gelir ve çalıştırılırlar. View'ler ise sorgulanırlar. View'lerin normal **SELECT** cümlelerinden farkı yoktur. Hatta aksine basit bir tek tablo sorgusu içeren view, içerisindeki **SELECT** sorgusuna göre daha yavaş çalışır.

- View'lerin parametrik olanına ihtiyacınız varsa, doğru seçim **Kullanıcı Tanımlı Fonksiyonlar** olabilir.

## VIEW OLUŞTURMAK

View'lerin en temel kullanımını inceleyelim. Aşağıdaki sorgu yapısı genel olarak kullanılan ve genelde geliştirme aşamasında işinizi görecek yapıdır.

---

```
CREATE VIEW view_ismi
AS
Sorgu_ifadeleri
```

---

View'lerden daha etkin yararlanabilmek için diğer özelliklerini de kullanabilirsiniz.

---

```
CREATE VIEW [schema_ismi].view_ismi [sutun_isim_listesi]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
Sorgu_ifadeleri
WITH CHECK OPTION
```

---

**Production.Product** tablosundaki veriler için bir view tanımlayalım. Bu view'de ürünlerin (*product*) temel bilgileri olan 4 sütun olsun.

---

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

---

Bu sorgu ile oluşturduğumuz view'i çalıştırarak sonuçlarına bakalım.

---

```
SELECT * FROM vw_Urunler;
```

---

	ProductID	Name	ProductNumber	ListPrice
1	1	Adjustable Race	AR-5381	0,00
2	2	Bearing Ball	BA-8327	0,00
3	3	BB Ball Bearing	BE-2349	0,00
4	4	Headset Ball Bearings	BE-2908	0,00

500 civarında kayıt listelenmiştir.

Göründüğü gibi bu view artık bir tablo gibi sorgulanabilir. İsterseniz yıldız (\*) işaretiyile değil, sütun isimlerini yazarak da sorguya kapsam sınırlaması uygulayabilirsiniz.

---

```
SELECT ProductID, Name, ProductNumber, ListPrice FROM vw_Urunler;
```

---

Bu sorgu ile tabloyu sorgulamak arasında veri gösterimi açısından bir fark yoktur. Bir view sorgularken kapsam sınırlaması yapabileceğiniz gibi **WHERE** filtreleme komutu gibi diğer **SELECT** sorgularında kullandığınız SQL komutlarını da kullanabilirsiniz.

View'lerde sütunlara isim vermeye ve veri tipi belirtmeye gerek yoktur. View içerisindeki **SELECT** sorgusu ile birlikte gelen sütunların ismi ve veri tipi kullanılacaktır.

View işleminde gerçekleşen asıl olay, basit ya da karmaşık bir sorgunun sanal bir isimle kısa ve daha anlaşılır, kolay kullanılabilir hale getirilmesidir. Bu işlemin doğru zamanda, doğru yerde kullanılması gereklidir. Çünkü normal bir **SELECT** sorgusunda sorgulama işlemi ve sorgunun karşısındaki verinin elde edilmesi işlemi gerçekleşir. Ancak view kullanımında bu sorgulama sürecini uzatacak ek katman oluşturulur. Öncelikle view yapısı metadata bilgileriyle ayrıştırılır, daha sonra elde edilen SQL sorgusu veritabanına iletilerek sorgu sonucu alınır. View sorgularında kullandığınız **SELECT** sorgusu, view'in kendisinden daha hızlı ve performanslı çalışacaktır. Bu nedenle özel gereksinim duymuyorsanız view kullanımından kaçınmalısınız.

## KISITLAMALAR

Birçok veritabanı nesnesinde olduğu gibi view'lerin da bazı kısıtlama kuralları vardır. Bunlar;

- View oluşturmak için kullanılacak sorgularda en fazla 1024 sütun (kolon) seçilebilir.
- View'ler geçici tabloları base tablo olarak kullanamazlar.

```
CREATE VIEW vw_hataliView
AS
SELECT sutun1, sutun2
FROM #geciciTablo
```

- View içerisindeki **SELECT** ifadesi, **ORDER BY**, **COMPUTE**, **INTO**, **OPTION** ya da **COMPUTE BY** yan cümleciklerini alamaz.

```
CREATE VIEW vw_hataliView
AS
SELECT sutun1, sutun2
FROM tablo_ismi
COMPUTE SUM(sutun1) BY sutun2
```

## GELİŞMİŞ SORGULAR İLE VIEW KULLANIMI

View konusunun başında kullanım sebeplerini sayarken bahsettiğimiz "kod karmaşıklığını azaltmak" özelliği, sanıyorum view kullanımının en çok tercih edilme sebebidir. Uzun kod bloklarını daha kısa ve kolay okunabilir hale getirmek için view ideal bir yöntemdir.

Karmaşık view sorgularında genellikle **JOIN** işlemleri için kullanılır. Bir ya da daha fazla tablodan birçok birleştirme işlemi gerçekleştirilerek elde edilen veriler tek bir kayıt kümesi olarak görüntülenmek istenir. Ancak her seferinde belki onlarca satır SQL kodu içeren **JOIN** sorgularını tekrar yazmak ya da çalıştırmak, uygulama içerisinde kullanmak pek kullanışlı bir yöntem değildir. İşte bu sorunu ortadan kaldırmak için view kullanılabılır.

`Production.Product` tablosunu kullanarak bir örnek yapalım. Bu tablodaki ürünlerden en çok istenebilecek istatistik sorgusu şu olabilir.

`Production.Product` tablomuzda hangi ürünler için kaç adet sipariş olduğunu, her siparişte kaç adet satıldığını ve elde edilen geliri gösteren sorgu hazırlayalım.

---

```
USE AdventureWorks
GO
CREATE VIEW vw_MusteriSiparisler
AS
SELECT
    soh.SalesOrderID,
    soh.OrderDate,
    sod.ProductID,
    p.Name AS UrunAd,
    sod.OrderQty,
    sod.UnitPrice AS BirimFiyat,
    sod.LineTotal AS SatirToplam
FROM
    Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID;
```

---

Şimdi hazırladığımız view'i çalıştıralım.

---

```
SELECT * FROM vw_MusteriSiparisler;
```

---

	SalesOrderID	OrderDate	ProductID	UrunAd	OrderQty	BirimFiyat	SatirToplam
1	43659	2005-07-01 00:00:00.000	709	Mountain Bike Socks, M	6	5,70	34.200000
2	43659	2005-07-01 00:00:00.000	711	Sport-100 Helmet, Blue	4	20,1865	80.746000
3	43659	2005-07-01 00:00:00.000	712	AWC Logo Cap	2	5,1865	10.373000
4	43659	2005-07-01 00:00:00.000	714	Long-Sleeve Logo Jersey, M	3	28,8404	86.521200
5	43659	2005-07-01 00:00:00.000	716	Long-Sleeve Logo Jersey, XL	1	28,8404	28.840400
6	43659	2005-07-01 00:00:00.000	771	Mountain-100 Silver, 38	1	2039,994	2039.994000
7	43659	2005-07-01 00:00:00.000	772	Mountain-100 Silver, 42	1	2039,994	2039.994000

Bu sorgumuz ile istenilen küçük rapor isteğini gerçekleştirdik ve sonuç olarak da 121.000 üzerinde kayıt listelendi. Bu ve daha gelişmiş sorgulardaki gibi

kodlaması uzun sürecek sorguları gelen isteklere göre hazırlayıp son kullanıcıya sadece view isimlerinden oluşan liste vererek, bu listedeki view'lar ile çalışıp istediği sonuçları hızlıca almasını sağlayabiliriz.

Tablo şeklinde sorgulayabildiğimiz müşteri siparişleri view'ı üzerinde **WHERE** ile filtreleme sorguları da oluşturulabilir.

---

```
SELECT UrunAd, BirimFiyat, SatirToplam
FROM vw_MusteriSiparisler
WHERE OrderDate = '2005-07-01';
```

---

	UrunAd	BirimFiyat	SatirToplam
1	AWC Logo Cap	5,1865	5.186500
2	AWC Logo Cap	5,1865	31.119000
3	AWC Logo Cap	5,1865	10.373000
4	AWC Logo Cap	5,1865	15.559500
5	AWC Logo Cap	5,1865	10.373000
6	AWC Logo Cap	5,1865	15.559500
7	AWC Logo Cap	5,1865	20.746000

## TANIMLANAN VIEW'LERİ GÖRMEK VE SİSTEM VIEW'LERİ

Bir veritabanında tanımlanmış tüm view'leri sistem view'leri ile listeleyebiliriz ve sistem view ya da şemaları master veritabanında bulunur.

View'ler hakkında bilgi içeren sistem şemalarından bazıları;

- **information\_schema.tables**

Veritabanında tanımlı view'lerin listesini içerir. Base tablosu olarak sysobjects'i kullanır.

- **information\_schema.view\_table\_usage**

Hangi tablolar üzerinde view tanımlı olduğunu gösterir. sysdepends tablosunu kullanır.

- **information\_schema.views**

Tanımlı view'lerin adı, kaynak kodu gibi tanımlama özelliklerini tutar. syscomments ve sysobjects'ten veri çeker.

Veritabanındaki tüm view'leri listeleyelim.

---

```
SELECT *
FROM sys.views;
```

---

Veritabanındaki tüm view'leri listelemek için JOIN yapısını kullanalım.

---

```
SELECT
    s.Name AS SchemaName,
    v.Name AS ViewName
FROM
    sys.views AS v
    INNER JOIN sys.schemas AS s
        ON v.schema_id = s.schema_id
ORDER BY s.Name, v.Name;
```

---

	SchemaName	ViewName
1	dbo	CustomerOrders_vw
2	dbo	pw_ManagementDepartment
3	dbo	pw_ManagementDepartmentProducts
4	dbo	vw_MusteriSiparisler
5	dbo	vw_Urunler
6	HumanResources	vEmployee
7	HumanResources	vEmployeeDepartment

Bütün view'lerin kaynak kodlarını görmek için;

---

```
SELECT *
FROM INFORMATION_SCHEMA.VIEWS;
```

---

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	VIEW_DEFINITION
1 AdventureWorks2012	Sales	vStoreWithContacts	CREATE VIEW [Sales].[vStoreWithContacts] AS SELECT ...
2 AdventureWorks2012	Sales	vStoreWithAddresses	CREATE VIEW [Sales].[vStoreWithAddresses] AS SELEC...
3 AdventureWorks2012	Purchasing	vVendorWithContacts	CREATE VIEW [Purchasing].[vVendorWithContacts] AS S...
4 AdventureWorks2012	Purchasing	vVendorWithAddresses	CREATE VIEW [Purchasing].[vVendorWithAddresses] AS ...
5 AdventureWorks2012	dbo	vw_Urunler	CREATE VIEW vw_Urunler AS SELECT ProductID, Nam...
6 AdventureWorks2012	dbo	pw_ManagementDepartment	CREATE VIEW pw_ManagementDepartment AS SELECT * ...
7 AdventureWorks2012	dbo	pw_ManagementDepartmentProducts	CREATE VIEW pw_ManagementDepartmentProducts AS ...

Her view'e ait sütunları listeleyelim.

---

```
SELECT
    v.Name AS ViewName,
    c.Name AS ColumnName
FROM
    sys.columns AS c
    INNER JOIN sys.views AS v
    ON c.object_id = v.object_id
ORDER BY
    v.Name, c.Name;
```

---

	ViewName	ColumnName
1	CustomerOrders_vw	LineTotal
2	CustomerOrders_vw	Name
3	CustomerOrders_vw	OrderDate
4	CustomerOrders_vw	OrderQty
5	CustomerOrders_vw	ProductID
6	CustomerOrders_vw	SalesOrderID
7	CustomerOrders_vw	UnitPrice

## VIEW'LERİN YAPISINI GÖRÜNTÜLEMEK

Sahip olduğumuz view'lerin hepsini kendimiz geliştirmiş olmayı bilmiyoruz. Ya da uzun süre önce geliştirdiğimiz view'lerin tanımlama bilgilerini hatırlamıyor olabiliriz. Bu durumda view'lerin içeriklerini yani yapılarını görüntülememiz gereklidir. Bunun için SSMS kullanılabilir. Bunun için aşağıdaki yolu izleyebilirsiniz.

Object Explorer -> Databases -> Veritabani\_Ismini -> Views -> [SağKlik] -> Design

Ancak her zaman SSMS ya da herhangi bir geliştirme aracına sahip olamayabiliriz. Bir geliştirici olarak bunun SQL tarafından nasıl yapıldığı bilinmelidir.

Bunun için SQL Server'da en çok kullanılan yöntemleri inceleyeceğiz.

### SYS.SQL\_MODULES

Mevcut view'e ait T-SQL kodlarını `sys.sql_modules` sistem kataloğu ile görüntüleyebiliriz.

---

```
SELECT Definition
FROM sys.sql_modules
WHERE Object_ID = OBJECT_ID('vw_MusteriSiparisler');
```

---

Definition
1 CREATE VIEW vw_MusteriSiparisler WITH SCHEMABINDING AS SELECT soh....

Bu sorgu sonucunda `vw_MusteriSiparisler` isimli view'e ait kodlar görüntülenir.

## **OBJECT\_DEFINITION**

Aynı işlemi `Object_Definition` fonksiyonunu kullanarak da gerçekleştirebiliriz.

---

```
SELECT OBJECT_DEFINITION(OBJECT_ID('vw_MusteriSiparisler'));
```

---

(No column name)
1 CREATE VIEW vw_MusteriSiparisler WITH SCHEMABINDING AS SELECT soh.Sal...

## **SYS.SYSCOMMENTS**

`syscomments` tablosunda `object ID`'ler üzerinden bu işlemi gerçekleştirmemizi sağlar. `object ID` özelliği SQL Server'ın nesneleri izlemek için nesnelere atadığı tam sayılı değerlerdir. `Object ID`'nin detaylarına girmeyeceğiz ancak bu işlem için kullanımı şu şekildedir.

---

```
SELECT
    sc.text
FROM sys.syscomments sc
    JOIN sys.objects so
        ON sc.id = so.object_id
    JOIN sys.schemas ss
        ON so.schema_id = ss.schema_id
WHERE so.name = 'vw_MusteriSiparisler' AND ss.name = 'dbo';
```

---

text
1 CREATE VIEW vw_MusteriSiparisler WITH SCHEMABINDING AS SELECT soh.Sales...

Bu sorguda `vw_MusteriSiparisler` view'ini arıyoruz. `ss.name` kısmında belirttiğim dbo ise, bu nesnenin hangi şema içerisinde ise o şemanın adı olmalıdır. Ben `vw_MusteriSiparisler` view nesnesini dbo schema (şema)'sında oluşturduğum için dbo kullandım. Bildiğiniz gibi SQL Server'da önceden tanımlanmış komutlar ve sistem prosedürleri de arka planda aslında yukarıdaki gibi T-SQL sorguları kullanırlar.

### **SP\_HELPTEXT**

Değişiklikler meydana geldiği anda sistem tablolarındaki güncel değişiklikleri gösteren `sp_helptext` komutunu da kullanabiliriz.

---

```
EXEC sp_helptext 'vw_MusteriSiparisler';
```

---

`sp_helptext` ile alacağımız sorgu sonucu diğerleri gibi tek satır halinde değil, T-SQL kodunu geliştirirken oluşturulan satır sayısı ve formatında görüntülenecektir. `sp_helptext` komutu metinsel parametre aldığı için view ismini tek tırnaklar içerisinde belirtmelisiniz.

`sp_helptext` komutu da aslında `syscomments` kısmında hazırladığımız sorguyu kullanır. Her şey bu şekilde geliştiricilerin işini hızlandırmak ve kolaylaştırmak, daha az kod yazılmasını sağlamak için hazır komutlar haline getirilir.

Text
1 CREATE VIEW vw_MusteriSiparisler
2 WITH SCHEMABINDING
3 AS
4 SELECT
5 soh.SalesOrderID,
6 soh.OrderDate,
7 sod.ProductID,
8 p.Name AS UrunAd,
9 sod.OrderQty,
10 sod.UnitPrice AS BirimFiyat,
11 sod.LineTotal AS SatirToplam
12 FROM
13 Sales.SalesOrderHeader AS soh
14 JOIN Sales.SalesOrderDetail AS sod
15 ON soh.SalesOrderID = sod.Sales...
16 JOIN Production.Product AS p
17 ON sod.ProductID = p.ProductID;

## **T-SQL İLE VIEW ÜZERİNDE DEĞİŞİKLİK YAPMAK**

Bir view üzerinde değişiklik yapmak için `ALTER` deyimi kullanılır.

---

```
ALTER VIEW [schema_ismi].view_ismi [sutun_isim_listesi]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
Sorgu_ifadeleri
WITH CHECK OPTION
```

---

Bir view oluşturmak ile değiştirmek arasındaki tek fark; **CREATE** yerine **ALTER** deyiminin kullanılmasıdır. **ALTER** ile view'e seçenek eklemek için **WITH** komutundan sonra **ENCRYPTION**, **SCHEMABINDING** komutları kullanılabilir.

## VIEW TANIMLAMALARINI YENİLEMEK

SQL Server'da view ile ilgili bilgiler metadata olarak tutulur. Bu bilgiler anlık olarak değil, belli periyodlar ile güncellenir. Örneğin; view'de kullanılan tablo ya da tabloların sütunlarında yapısal bir değişiklik olduğunda bu değişiklikler arasında metadata bilgilerine yansımayabilir. Metadata'ya yansımayan güncellemeler view tarafından bilmemektedir. Bunu işlemi manuel olarak biz şu şekillerde gerçekleştirebiliriz.

---

```
EXEC sp_refreshview 'vw_MusteriSiparisler';
```

---

Aynı işlemi **sp\_refreshsqlmodule** prosedürünu kullanarak da gerçekleştirebiliriz.

---

```
EXEC sp_refreshsqlmodule @name = 'vw_MusteriSiparisler';
```

---

## KOD GÜVENLİĞİ: VIEW'LERİ ŞİFRELEMEMEK

Geliştirilen uygulamalarda genel olarak veritabanları ve sorguları erişime açık ortamlarda çalışmak zorundadır. Bu gibi durumlarda veritabanı ve sorguların güvenliğinin önemi artmaktadır. Uygulamanın güvenliği ve bazı ticari kaygılarından dolayı, geliştirilen SQL kodlarının içeriği veritabanı katmanında güvenlik altına alınması gerekebilir. View, içerisindeki kodları şifreleyerek kodlarını gizleme ve koruma imkanı sağlar. Bu özellik kullanılmadan önce şifrelenecek SQL sorgularının yedeği alınmalıdır. Çünkü şifreleme işleminden sonra geliştirici dahil hiç kimse şifrelenen sorguları görüntüleyemez.

View'leri şifrelemek için **ENCRYPTION** operatörü kullanılır. Şifreleme işlemi view oluşturulurken (**CREATE**) belirtilebileceği gibi sonradan güncelleme (**ALTER**) işlemiyle de belirtilebilir.

---

Şifreli bir view oluşturmak için;

---

```
CREATE VIEW [schema_ismi].view_ismi [sutun_isim_listesi]
WITH ENCRYPTION
AS
Sorgu_ifadeleri
```

---

Mevcut bir view'i şifreli hale getirmek için ise **CREATE** yerine **ALTER** kullanmanız yeterli olacaktır.

View güncelleme konusunda belirttiğimiz bir konuya şifreleme konusunda hatırlamalıyız. Bir view'i **ALTER** ile güncellerken her güncelleme işleminde **WITH ENCRYPTION** ile tekrar şifreli view oluşturmak istediğiniz bildirmelisiniz. Bu bildirimi yapmazsanız SQL Server bu durumu "*şifreleme işleminden vazgeçildi*" şeklinde yorumlayarak view tanımlamalarını şifresiz olarak saklayacaktır.

Bir önceki örnekte tanımladığımız **vw\_MusteriSiparisler** view'in tanımını gizleyelim.

---

```
ALTER VIEW vw_MusteriSiparisler
WITH ENCRYPTION
AS
SELECT
    soh.SalesOrderID, soh.OrderDate, sod.ProductID,
    p.Name AS UrunAd, sod.OrderQty,
    sod.UnitPrice AS BirimFiyat, sod.LineTotal AS SatirToplam
FROM
    Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
    ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
    ON sod.ProductID = p.ProductID;
```

---

Şifreleme işlemini kontrol edelim.

---

```
sp_helptext 'vw_MusteriSiparisler';
```

---

The text for object 'vw\_MusteriSiparisler' is encrypted.

İçerğini görüntülemek istediğimiz view'in encrypted (*şifreli*) olduğunu belirten uyarı aldık.

## SCHEMA BINDING

Bu özelliği ile view'in bağlı olduğu tablo ya da diğer view'ler gibi nesneleri istenen view'e bağlar. **Schema binding** ile bağlanan view'lerin şema bağlantısı kaldırılmadan üzerinden `Create`, `Alter` gibi bir değişiklik yapılamaz.

**Schema Binding** işlemine neden ihtiyaç duyuyoruz?

- Bir ya da birden fazla tablomuzla ilişkili bir view oluşturduğumuzda başka bir geliştiricinin ya da unutarak kendi hazırladığımız view'in yapısını bozacak (view içindeki bir sütunun `DROP` edilmesi gibi), sorgunun çalışmasında hatalar oluşturacak şekilde ilgili tablolarda değişiklikler yapmamızı engeller.
- View'i referans alan bir schema bağlantılı kullanıcı tanımlı fonksiyon oluşturacaksanız view'da schema bağlantılı olmak zorundadır.
- View'da index kullanılması gereken durumlar olabilir. Bu durumda view'de indeks kullanılmak için view'i oluştururken `SCHEMABINDING` özelliği kullanılmak zorundadır.

## VIEW İLE VERİLERİ DÜZENLEMELİ:

### INSERT, UPDATE, DELETE

Veritabanında herhangi bir tablo üzerinde gerçekleştirdiğimiz `Insert`, `Update`, `Delete` işlemlerini bir tablonun sanal görünümü olan view'ler üzerinden de gerçekleştirebiliriz. Tabii ki view'in yapısından kaynaklanan bazı kısıtlamalar ile birlikte bu işlem gerçekleştirilebilir.

View üzerinden `Insert`, `Update`, `Delete` işlemleri gerçekleştirmenin kısıtlamaları

- Kümeleme fonksiyonları, `GROUP BY`, `DISTINCT` ve `HAVING` kullanılamaz.
- Insert ile veri eklenecek view'de ilişkili bir tablonun Identity sütununa veri eklenemez. Identity otomatiktir.
- Insert gerçekleştirilecek view'de hesaplanmış sütunlara veri eklenemez.
- Birden fazla tablodan veri çeken view'da gerçekleştirilecek ise `Instead Of Trigger` kullanılmalıdır.
- Bir view'in base tablosu üzerinde değişiklik yapabilmesi için `Constraint`, `Unique Index`'lere takılmaması gereklidir. Base tabloda `NULL` olamayan sütunlar varsa ve

bu sütunlar view'de yer almıyorsa Insert işlemi sırasında bu **NULL** geçilemeyen alanlar **NULL** geçilmiş olacağı için hata meydana gelecektir.

- Kullanıcının view üzerinden erişimi olduğu ancak view sorgusu sonucunda eklediği kaydı göremeyeceği durumlarda kullanıcının veri eklemesi doğru bir yaklaşım olmayacağıdır. Bu gibi durumları önlemek için **CHECK OPTION** operatörü kullanılmalıdır.

**Production.Location** tablosu üzerinde bir view oluşturalım.

---

```
CREATE VIEW Production.vw_Location
AS
SELECT LocationID, Name,
       CostRate, Availability
FROM Production.Location;
```

---

Oluşturduğumuz view'e veri ekleyelim.

---

```
INSERT INTO Production.vw_Location(Name, CostRate, Availability)
VALUES('Dijibil Test',1.10,50.00);
```

---

Hazırladığımız **vw\_Location** isimli view'de veri eklemeyi engellemeyecek herhangi bir engel bulunmadığı için bu veri ekleme işlemi sorunsuz bir şekilde gerçekleşecektir.

Eklediğimiz kaydı listeleyelim.

---

```
SELECT *
FROM Production.vw_Location
WHERE Name = 'Dijibil Test';
```

---

	LocationID	Name	CostRate	Availability
1	61	Dijibil Test	1.10	50.00

## VIEW İLE VERİLERİ DÜZENLEMEDE INSTEAD OF TRIGGER İLİŞKİSİ

View'ler tek tablo kaynak olarak kullanıldığından ve karmaşık sorgular içermediği taktirde bir tablodan farklı değildir. Bu tür view'lerde veri ekleme, güncelleme

ve silme işlemlerini gerçekleştirebilirsiniz. Ancak karmaşık sorgular içeren, **JOIN** ile birden fazla tabloyu birlestiren sorgular kullanıldığında bu işlemleri gerçekleştirmek bazı şart ve zorluklara tabi tutulur.

- View, birden fazla tablonun birleşimini içeriyorsa, birçok durumda **Instead Of Trigger** kullanmadan veri ekleme (**Insert**) ya da silme (**Delete**) işlemini gerçekleştiremezsiniz. Veri güncelleme işleminde ise, tek tabloyu base tablo olarak kullanmak şartıyla Instead Of Trigger kullanmadan güncelleme gerçekleştirilebilir.
- Eğer view sadece bir tabloya bağlı ise ve view veri eklenecek tüm sütunları içeriyorsa ya da sütunlar **DEFAULT** değere sahipse, veri ekleme işlemini, view üzerinden Instead Of Trigger kullanmadan gerçekleştirilebilir.
- View tek bir tabloya bağlı olsa da veri eklenecek sütun view'de gösterilmemişse ve **DEFAULT** değere de sahip değilse, bu durumda, view üzerinden veri eklemek için Instead Of Trigger kullanmak zorundasınız.

Trigger konusuna derinlemesine giriş yapmadığımız bu konuda Instead Of Trigger konusunu içermesinin sebebi; view ile trigger'lar arasındaki bağ ve zorunlulukları kavrayabilmenizdir. Özetlemek gerekirse; Instead Of Trigger, hangi işlem için tanımlandıysa onun yerine çalışan bir trigger türüdür. Bu trigger işlemi sonucunda işlemin ne yapacağını görebilir ve oluşacak herhangi bir sorunu çözmek için trigger'da önlem alınabilir.

## **JOIN İŞLEMİ OLAN VIEW'LERDE VERİ DÜZENLEMEK**

**JOIN** sorguları içeren view'lerde veri düzenleme işlemi gerçekleştirmek için Instead Of Trigger'lar kullanılmak zorundadır. **JOIN**, yapısı gereği birden fazla tabloda birden fazla sütun ve sorgu içerir. Bu nedenle veri düzenleme işlemi için gereken bilgilerde hatalar olması kaçınılmazdır. Microsoft, bu karmaşanın önüne geçmek için, varsayılan olarak çoklu tablo içeren view'lerde veri düzenlemeye izin vermez. Ancak bu işlemi kullanmak bir gereklilik ise Instead Of Trigger kullanabilirisiniz.

## WITH CHECK OPTION KULLANIMI

Bir view ile oluşturululan **SELECT** sorgusu, base tabloyu görme yetkisi olmayan ancak kısıtlı bir veriye erişebilen bir view kullanma hakkı varsa ve bu yetkisiyle kullandığı view üzerinden veri ekleme işlemi gerçekleştirebilir. Soru başarıyla çalışabilir. Ancak bu eklediği veriyi bir daha göremez.

Bir başka deyişle, view kullanılarak eklenecek veri view'i oluşturan **SELECT** ve **WHERE** sorgusu sonucunda dönecek bir kayıt olmalıdır. Bu şartlara uymayan verinin eklenmesini engellemek için **WITH CHECK OPTION** kullanılır.

## İNDEKSLENMİŞ VIEW'LER

İndekslenmiş view, clustered indeksler biçiminde maddeleştirilmiş, **unique** (*benzersiz*) değerler setine sahip view'dır. Bunun avantajı, view ile birlikte view'in temsil ettiği bilgiyi de elde ederek, çok hızlı arama sağlamasıdır.

Clustered olmak zorunda olan ilk indeksten sonra, SQL Server ilk indeksin işaret ettiği clustered key değerini referans alarak, view üzerinde ek indeksler oluşturabilir.

View üzerinde indeks kullanabilmek için bazı kısıtlamalar

- View **SCHEMABINDING** özelliğini kullanmak zorundadır.
- View tarafından referans gösterilen her fonksiyon **belirleyici** (*deterministic*) olmak zorundadır.
- View, referans gösterilen tüm nesnelerle aynı veritabanında olmak zorundadır.
- Sadece tablolar ve **UDF** (*kullanıcı-tanımlı fonksiyonlar*)'lere referans olabilir. View başka bir view'e referans olamaz.
- View herhangi bir kullanıcı-tanımlı fonksiyona referans gösteriliyorsa, view'ler de schema bağlantılı olmalıdır.
- View'de referans gösterilen tüm tablolar ve UDF'ler iki parçalı (`dbo.Production` gibi) isimlendirme kuralına uymak zorundadır. Ayrıca view ile sahibinin de aynı olması gereklidir.
- View ve view'i oluşturan tüm tablolar oluşturulurken, **ANSI\_NULLS** ve **QUOTED\_IDENTIFIER** özellikleri aktif olmalıdır. Eğer aktif değil ise **SET** komutu ile bunu yapabilirsiniz.

**Örnek:**

**SET ANSI\_NULLS ON** ya da **SET ANSI\_NULLS OFF** ile bu özellik aktif ya da pasif edilebilir.

View bölümünün başında hazırladığımız **vw\_MusteriSiparisler** view'ini **ALTER** ile yeniden düzenleyerek indeks kullanımına hazır hale getiriyoruz.

---

```
ALTER VIEW vw_MusteriSiparisler
WITH SCHEMABINDING
AS
SELECT
    soh.SalesOrderID,
    soh.OrderDate,
    sod.ProductID,
    p.Name AS UrunAd,
    sod.OrderQty,
    sod.UnitPrice AS BirimFiyat,
    sod.LineTotal AS SatirToplam
FROM
    Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
    ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
    ON sod.ProductID = p.ProductID;
```

---

Bu view'i daha önce oluşturmadıysanız **ALTER** yerine **CREATE** komutu kullanarak, yeni bir view oluşturabilirsiniz.

Burada dikkat edilmesi gereken bir diğer özellik de view'i **WITH SCHEMABINDING** ile oluşturmuş olmamızdır. **SCHEMABINDING** indekslenmiş view'ler için bir gerekliliktr.

Şimdiye kadar yaptıklarımızla indeksli bir view'e henüz sahip olmasak da view'imizi indekslenebilir hale getirdik. Şimdi bu view'i kullanarak yeni bir indeks oluşturalım. İlk oluşturacağımız indeks hem unique hem de clustered olmak zorundadır.

---

```
CREATE UNIQUE CLUSTERED INDEX indexedvMusteriSiparisler
ON vw_MusteriSiparisler(SalesOrderID, ProductID, UrunAd);
```

---

Hazırladığımız view ve indeks sorgumuza dikkat ederseniz 3. parametreyi **UrunAd** olarak belirttiğim. Bunu fark edebilmeniz için bilinçli olarak kullandım. Diğer **SalesOrderID** ve **ProductID** gerçek sütun isimleriyle UrunAd bir takma isim yani sanal bir isimdir. Ancak sorgu içerisinde bu isimlendirme ile belirttiğimiz için indeks içerisinde de bu ismi kullanmanız gereklidir.

```
vw_MusteriSiparisler(SalesOrderID, ProductID, UrunAd)
```

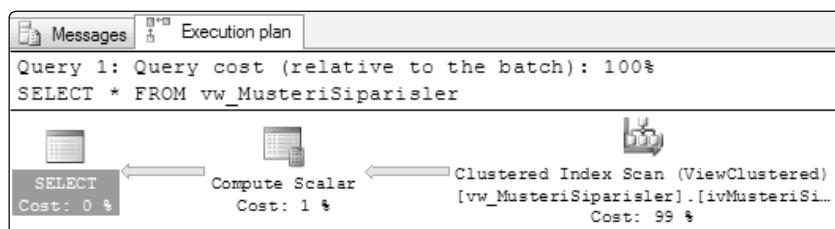
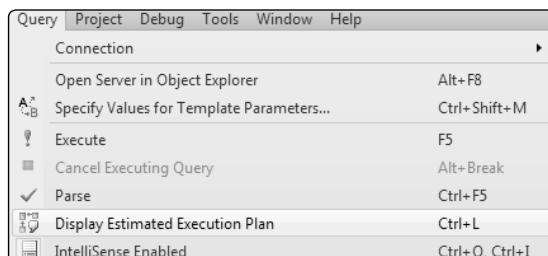
**UrunAd** yerine **Name** yazarak çalıştırıldığınızda sorgu hata verecektir.

View'i çalıştırarak sorgu sonucunu inceleyelim.

```
SELECT * FROM vw_MusteriSiparisler;
```

	SalesOrderID	OrderDate	ProductID	UrunAd	OrderQty	BirimFiyat	SatırToplam
1	43659	2005-07-01 00:00:00.000	709	Mountain Bike Socks, M	6	5,70	34.200000
2	43659	2005-07-01 00:00:00.000	711	Sport-100 Helmet, Blue	4	20,1865	80.746000
3	43659	2005-07-01 00:00:00.000	712	AWC Logo Cap	2	5,1865	10.373000
4	43659	2005-07-01 00:00:00.000	714	Long-Sleeve Logo Jersey, M	3	28,8404	86.521200
5	43659	2005-07-01 00:00:00.000	716	Long-Sleeve Logo Jersey, XL	1	28,8404	28.840400
6	43659	2005-07-01 00:00:00.000	771	Mountain-100 Silver, 38	1	2039,994	2039.994000
7	43659	2005-07-01 00:00:00.000	772	Mountain-100 Silver, 42	1	2039,994	2039.994000

View'i çalıştırıldıktan sonra **SSMS**'de **Query** menüsünde **Display Estimated Execution Plan** menüsünü kullanarak hazırladığımız view'e ait **Execution Plan'a** ulaşabiliriz.



## PARÇALI VIEW KULLANIMI

Parçalı view'ler, aynı yapılara sahip (eşteş) birden fazla tablodaki verilerin sonuçlarını birleştirerek tek kayıt kümesi olarak gösteren view'lerdir. Parçalı view'lerin birkaç farklı kullanım alanı mevcuttur. Örneğin; birden fazla ve farklı veri kaynağından alınan verileri birleştirerek tek bir tablo gibi görüntülenmesini sağlar.

Birden fazla şubesi bulunan bir firmanın, merkez şubesinde bulunan yönetim departmanı diğer şubelerin sipariş bilgilerini görüntüleyebilmek için parçalı view iyi bir çözüm olacaktır. Ayrıca bu şubelerin farklı veritabanlarını kullanıyor olmaları parçalı view kullanımını engelmez. Bir şube SQL Server, diğeri Oracle ya da bir başka veritabanı yazılımı kullanıyor olabilir.

Parçalı view yapısının kullanıldığı bir diğer alan ise, yüksek veri yiğinlarının bulunduğu tabloların belli kriterlere göre farklı tablolara ayrılabilmesini sağlamaktır. Örneğin; milyonlarca kayıt içeren bir tablo üzerinde sorgulama, raporlama işlemlerinin gerçekleştirilmesi uzun zaman ve kaynak tüketimine sebep olacağı gibi donanımsal anlamda da yönetimi zor olacaktır. Bu tür sorunların çözümünde parçalı view etkin olarak kullanılabilir.

Parçalı view oluşturmak için kullanılan genel söz dizimi;

---

```
CREATE VIEW view_ismi
WITH seçenekler
AS
Sorgu_ifadeleri
UNION ALL
SELECT ifadesi2
...
```

---

Birden fazla şubesи bulunan ve merkez şube tarafından, şube siparişlerinin takip edildiği bir sipariş takip sistemi geliştiriyoruz. Bu sistemin birden fazla kaynaktan veri almasını sağlayacak sorguyu hazırlayalım.

Öncelikle şubelerimizi oluşturup gerekli kayıtları girmeliyiz.

---

```
CREATE TABLE Department1Product
(
    DepartmentID TINYINT NOT NULL,
    ProductID INT,
    ProductName VARCHAR(50),
    ProductPrice MONEY,
    PRIMARY KEY(DepartmentID, ProductID),
    CHECK(DepartmentID = 1)
);
GO
INSERT INTO Department1Product
    (DepartmentID, ProductID, ProductName, ProductPrice)
VALUES
    (1,1,'İleri Seviye SQL Server', 50),
    (1,2,'İleri Seviye Oracle', 50);
```

---

Kayıt ekleme (**Insert**) işleminde, SQL Server özelliklerinden **multiple insert** (**çoklu ekleme**) özelliğini kullandık.

Şubemizde ürünlerle ilgili **ProductID**, **ProductName**, **ProductPrice** bilgilerinin ve ek olarak **DepartmentID** sütunumuz var. **DepartmentID** sütunu bizim parçalı yapı oluşturmayı sağlayan sütundur. Şubelerimizi ayrı olarak sorulayabilmek için **DepartmentID** sütununu kullanacağız.

Şimdi diğer şubemizi de tanımlayıp örnek kayıtlar ekleyelim.

---

```
CREATE TABLE Department2Product
(
    DepartmentID TINYINT NOT NULL,
    ProductID INT,
    ProductName VARCHAR(50),
    ProductPrice MONEY,
    PRIMARY KEY(DepartmentID, ProductID),
    CHECK(DepartmentID = 2)
);
GO
INSERT INTO Department2Product
    (DepartmentID, ProductID, ProductName, ProductPrice)
VALUES(2,1,'İleri Seviye Java', 50),
    (2,2,'İleri Seviye Android', 50);
```

---

Şubelerimizin tablolarını oluşturarak örnek kayıtlarımızı ekledik. Şimdi merkez şubemizden bu verileri görüntüleyebilmek, parçalı view yapısını tamamlayabilmek için gerekli view'i oluşturalım.

---

```
CREATE VIEW pw_ManagementDepartment_Product
AS
SELECT * FROM Department1Product
UNION ALL
SELECT * FROM Department2Product
```

---

Artık oluşturduğumuz **pw\_ManagementDepartmentProducts** view'i çalıştırarak tüm şubelerimizdeki kayıtları görüntüleyebiliriz.

---

```
SELECT * FROM pw_ManagementDepartment_Product;
```

---

	DepartmentID	ProductID	ProductName	ProductPrice
1	1	1	İleri Seviye SQL Server	50,00
2	1	2	İleri Seviye Oracle	50,00
3	2	1	İleri Seviye Java	50,00
4	2	2	İleri Seviye Android	50,00

Şubelerimizi oluşturarak merkez şubemizden ürünleri listelemeyi gerçekleştirdik. Dikkat ederseniz şuan sadece şubelerden kayıt girişi yapılabiliyor. Ancak gerçek uygulamada merkez şube bu tablolara müdahale ederek kayıt ekleyebilmelidir. Tablolara kayıt eklemek için oluşturduğumuz view'i kullanabiliriz.

---

```
INSERT INTO pw_ManagementDepartmentProducts
(DepartmentID, ProductID, ProductName, ProductPrice)
VALUES
(1,3,'İleri Seviye Robot Programlama','50'),
(1,4,'İleri Seviye Yapay Zeka','55'),
(2,3,'İleri Seviye PHP','40'),
(2,4,'İleri Seviye JSP','40');
```

---

Buraya kadar herhangi bir hata ve sorun ile karşılaşmadık. Peki, merkez şubeden veritabanında daha önceden var olan bir ürünü tekrar ekemeye çalışsaydık ne olurdu?

Merkez şube için oluşturduğumuz view'i kullanarak var olan bir kaydı tekrar eklemeye çalışalım.

---

```
INSERT INTO pw_ManagementDepartmentProducts
(DepartmentID, ProductID, ProductName, ProductPrice)
VALUES
(2,1,'İleri Seviye Java',50),
(2,2,'İleri Seviye Android',50);
```

---

Bu sorgu sonucunda şube tablolarındaki kısıtlamalardan dolayı, bu verinin var olduğunu ve tekrar eklenmeyeceğini belirten aşağıdakine benzer bir hata alırız.

Violation of PRIMARY KEY constraint 'PK\_Departme\_794757A37141AFB4'. Cannot insert duplicate key in object 'dbo.Department2Product'. The duplicate key value is (2, 1).

## **PARÇALI VIEW KULLANILIRKEN DİKKAT EDİLMESİ GEREKENLER**

- Parçalı view'de kullanılan tablolarda **IDENTITY** sütun bulunmamalıdır.
- **UNION ALL** ifadesi parçalı view için önemlidir. Bu nedenle parçalı view'lerde **UNION ALL** ifadesinin şartlarının sağlanması gereklidir. Yani birleştirilecek sonuçların alındığı tabloların alanlar tür ve sıralama uyumlu döndürülmelidir.
- Parçalı view'de kullanılacak tablo için **Ayırıcı Sütun** (*partitioning column*) bulunmalıdır. Örneğimizde kullandığımız **DepartmentID** gibi bir ayırıcı sütun çözüm olabilir. Ayırıcı sütun hesaplanan bir sütun olamaz. Kayıt işlemlerinin kontrolü için örneğimizde oluşturduğumuz gibi ayırıcı sütun bir **CHECK CONSTRAINT** tanımı ile denetlenmelidir. Örneğimizdeki son kayıt ekleme işlemindeki hatayı hatırlayın.
- Birden fazla bağlı sunucu üzerindeki tabloyu bir araya getirerek parçalı view oluşturulabilir. Ancak öncelikle bu tabloları barındıran dağıtık sunucuların SQL Server'da tanımlanmış olmalıdır.

## **VIEW'LARI KALDIRMAK**

Mevcut view'ları basit bir `DROP` sorgusu ile kaldırabiliriz (silmek).

---

```
DROP VIEW view_ismi
```

---

Müşteri siparişleri için hazırladığımız view'ı kaldıralım.

---

```
DROP VIEW vw_MusteriSiparisler;
```

---

## **SSMS İLE VIEW OLUŞTURMAK VE YÖNETİMİ (VIDEO)**

# İNDEKSLERLE ÇALIŞMAK

Bu bölüme kadar, veritabanı programlama adına birçok farklı özellik ve yetenek inceledik. Veritabanı tablolarını oluşturduk. Bu tablolara veriler eklemeyi, güncellemeyi, silmeyi inceledik. Verileri seçerken farklı teknik ve sorgulama tekniklerini inceledik. Ancak, SQL Server'ın bu verileri nasıl depoladığını ve verilere ulaşırken kullandığı yöntemlere degeinmedik. Veriler tablolarda tutulur ve seçme, güncelleme, silme gibi komutlar ile yönetilir. Bu işlemler T-SQL geliştiricileri tarafından gerçekleştirilir.

Temel işlemler gerçekleştiren bir T-SQL geliştirici için verinin nasıl depolandığı ve arka planda nasıl yönetildiği çok önemli değildir. Ancak, bir veritabanı yöneticisi ya da ileri seviye bir T-SQL programcısı için verinin nasıl depolandığı, SQL Server'ın bu verilere nasıl ulaştığı, performans için hangi algoritma ve teknikleri kullandığı çok önemlidir.

Hayatın bize öğrettiği bir gerçek vardır. Hangi yolun daha kestirme olduğunu bilmek için, tüm yolların özelliklerini ve teknik bilgilerine hakim olmalısın.

Bu bölümde, SQL Server'ın veriyi daha performanslı kullanabilmek ve sorgu sonuç hızını artırmak için kullandığı Indeks mimarisini inceleyeceğiz.

İndeks kavramı, veritabanı programlamaya yeni başlayanlar için anlaşılması kolay olmayan bir mimari kavramdır. Yeni başlayanlar için şunu söyleyebilirim ki; Indeks mimarisini anlamakta zorlanmak gayet normaldir.

Tüm ilişkisel veritabanı yönetim sistemlerinde var olan Indeks kavramını anlatmak için nesnelleştirmek en iyi yöntemdir. Indeksler gerçek hayatı birçok farklı şekilde karşımıza çıkan ve gerçek hayat sorunlarına bulduğumuz çözümlerden ibarettir. Eskiden, telefon kulübelerinde ve evlerde telefon rehberleri vardı. Bu rehberlerde tüm telefon numaraları yazardı ve çok kalın bir kitaptı diyebilirim. Bu rehberlerde kayıtlı on binlerce telefon numarası arasında istenen kişi ya da kurumun telefon numarasını bulmak oldukça zor ve zaman gerektiren bir ihti. Telefon rehberinde yüz bin telefon numarası olduğunu düşünelim. Bu telefon numaralarının belli kurallara göre değil de, karmaşık olarak yazıldığını ve bir telefon numarası aradığınızı hayal edin! Çok zor olmaz mıydı? Telefon rehberindeki numaraları daha hızlı bulmak için bir yol olmazıydı. Bu sorunu çözmek için telefon rehberini üreten firmalar telefonları kişi ya da kurumların isimlerinin baş harfine göre, alfabetik olarak sıralamayı ve harflere göre kategorilendirme yöntemini kullanmaya başladılar. Bu yöntem hem daha anlaşılabılır bir rehber oluşmasını sağladı hem de rehberden telefon arayan kişiler aradıklarını daha hızlı bulabilir oldular.

Veriye hızlı ulaşım sadece bilgisayar dünyasında değil, gerçek hayatı da kullanılan ve gerekli olan önemli bir gereksinimdir. Veritabanı ve Indeks mimarisini anlamak için bir diğer gerçek hayatı örneği kütüphane modelidir. Bir kütüphanede binlerce ya da on binlerce kitap olabilir. Bu kitaplar belirli algoritmala göre düzenlenmeli ve raflardaki yerini almalıdır. Aksi halde ne olur?

Bir kütüphane çalışanı, her yeni gelen kitabı, kütüphanede boş gördüğü raflara rastgele eklediğini düşünelim. 500 yıllık bir tarihi kitap ile yeni basılmış bir edebiyat romanının raflarda yan yana durması ne kadar doğrudur? Kütüphaneden kitap almaya gelen birisi, kütüphaneciden bir kitap istediğiinde, kütüphaneci bu kitabı bulabilmek için tüm kitapları taraması gereklidir. Ancak, kütüphanedeki her kitap bir düzene göre sıralanmış olsa, yazar isimlerine göre alfabetik olarak, edebiyat, tarih vb. şekilde kategorilendirilmiş ve yıllara göre sıralama yapılsaydı, istenen her kitap hızlı bir şekilde kolaylıkla bulunabilirdi. Bu kütüphanedeki yazar isimlerine göre, tarih ve kategorilere göre sıralama işleminin veritabanındaki karşılığı Indekslerdir. Indeksler, veriye hızlı, performanslı ve kolay ulaşmak için kullanılırlar.

## SQL SERVER DEPOLAMA

SQL Server'da indeks ve performans kavramlarının anlaşılabilmesi için öncelikle veri depolama mimarisi incelenmelidir. SQL Server'da veriler farklı katmanlar ve nesnelerin birbirlerinin hiyerarşik bir parçası olarak yönetilir. Hiyerarşinin en altındaki katman ile en üstteki katman arasında farklı birçok nesne ve katman vardır. Bu nesnelerin bazlarının üzerinde çalışabildiğimiz için anlaşılması kolaydır. Bazı nesneler ise doğrudan erişilemese de erişilebilir durumdayken, bir kısmı nesneler de SQL Server'in sistemi tarafını ilgilendirdiği için dışarıdan erişime tamamen kapalı ve gizlidir. SQL Server, herhangi bir performans ve iyileştirme yapmadan da performanslı çalışabilecek şekilde tasarılmıştır. Ancak, mimari açıdan geliştirilen yöntemler ve performans artırıcı nesneler ile SQL Server'dan daha yüksek verim alınabilir.

Bu tür iyileştirmelere, geliştiricinin hazırladığı tablo ve veri yapısını SQL Server'a doğru olarak tanitarak verinin daha performanslı şekilde yönetilmesi denilebilir.

Veri performansı artırıcı konulara deðinmeden önce, SQL Server'in veri depolama için kullandığı yöntem ve mimari katmanları inceleyelim.

### VERİTABANI

Birçok veritabanında kullanılan bu terim bazı büyük veritabanlarında farklı isimlendirilebilir. Veritabanı, aynı amaç için oluşturulan tablo, veri ve nesneleri kapsayıcı ana nesnedir. Fiziksel anlamda dosyalarda tutuluyor olsa da aslında fiziksel değil, mantıksal bir kavramdır.

SQL Server'da veriler, veritabanı bazında ele alınarak işlenir ve güvenliği sağlanır. Performans için gerekli tüm düzenlemeler de bu ana kapsayıcı nesnenin bir parçasıdır.

### DOSYA

SQL Server, içerisindeki tüm nesne ve verileriyle birlikte fiziksel, yani disk'te bulunan dosyalarda tutulur. Bu dosyalar temel olarak ikiye ayrılır.

- **Veri Dosyası (\*.mdf)**

Veri dosyası, birincil ve vazgeçilmez veritabanı dosyasıdır. Veritabanının tüm nesne ve verilerini içerisinde barındırır. Varsayılan olarak dosya uzantısı .mdf

olsa da bu bir zorunluluk değildir. Farklı uzantılarda olabilir ve sorunsuz çalışır. Ancak, önerilen tabi ki bu dosya uzantısı ile kullanılmasıdır.

Veri katmanı için yapılan iyileştirmeler ve nesneler bu dosya içerisinde saklanır.

#### • Log Dosyası (\*.ldf)

Veritabanının ikincil ve diğer bir dosyası log dosyalarıdır. Log dosyaları **.ldf** (*log data file*) uzantısına sahiptir. Veritabanındaki işlemler için önemli bir yere sahiptir. Log dosyası olmadan veritabanı işlem yapmaz. Birçok bölümde log dosyası üzerinde farklı bazı işlemler gerçekleştirmiştir ve birçok durumda log dosyalarının önemine değişmiştir.

Log dosyaları veritabanı işlemleri için işlem loglarını tutmakla görevlidir. Aslında bazı durumlarda veritabanının can simididir diyebiliriz.

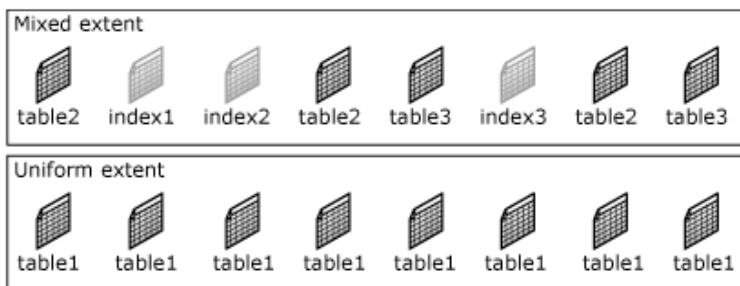
Veri ve log dosyaları birbirini tamamlayan bir bütününe iki parçasıdır. İkisi de gerekli dosyalardır. Büyük verilere sahip veritabanlarında .ndf gibi dosyalar da vardır. Bunlar, daha sonradan veritabanına eklenirler. Ancak temel anlamda en gerekli dosyalar veri ve log dosyalarıdır.

Büyük verileri yönetmek için tasarlanan profesyonel veritabanı uygulamalarında veri güvenliği ve performansı yüksek öneme sahiptir. Öyle ki, bazı durumlarda güvenliği artırmak için, performanstan ödün verilmesi gereken durumlar söz konusu olabilir. Performans ve güvenlik için temel olarak yapılan ilk işlemler gene bu dosyalar üzerinde gerçekleştirilir.

Bir sistemin disk'inin yazma ve okuma hızı standart olarak bellidir. Veri ve log dosyalarının her ikisinde de yazma ve okuma işlemi gerçekleştirilir. Her iki dosyanın da bir disk üzerinde tutulması hem performans hem de güvenlik açısından büyük bir risktir. Disk aynı anda hem ldf hem de mdf dosyalarında yazma ve okuma işlemi gerçekleştireceği için diskin yazma ve okuma hızı otomatik olarak yarı yarıya bölünmüş olacaktır. Bu duruma işletim sistemi ve diğer programların kullandığı yazma ve okuma işlemlerini de ekleyince, performans olumsuz anlamda etkilenecektir. Aynı zamanda, iki dosyanın aynı disk üzerinde tutulması güvenlik riskidir. Disk, sistemin çalışma anında herhangi bir sebepten dolayı ariza yaparsa veri ve log dosyası aynı anda kaybedilmiş olunur. Doğru olan, en azından işletim sistemi, veri dosyası ve log dosyası için ayrı ayrı, üç farklı disk kullanılması ve ayarların buna göre gerçekleştirilmemesidir.

## EXTENT

Tablolar ve Indeksler için kullanılan temel depolama birimidir. Yani, veritabanı içinde ayrılan birim alanıdır. Sekiz adet bitişik 64K'lık veri page'lerinden oluşur. Page kavramını detaylıca inceleyeceğiz.



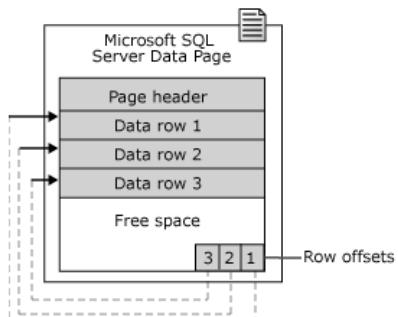
Extent dolduğunda, bir sonraki kayıt, kayıt boyutu kadar yeni bir extent'te yer alır. SQL Server extent'leri belirli durumların gerçekleşmesiyle otomatik olarak ayırrı. Bu durumda her extent dolduğunda yeni bir extent oluşturmak yerine, daha önceden oluşturulmuş bir extent'e geçilir. Bu nedenle herhangi bir bekleme olmaz. Diğer extent'ler sıranın kendisine gelmesini bekler.

## PAGE

Page'ler, extent'ler içerisinde bulunan, extent'lerin birim alanlarıdır. Her extent içerisinde 8 page bulunur. Her extent'teki page sayısı aynıdır. Yani her extent sekizlik sistem ile çalışır.

Extent'ler yapısal olarak belirli bir kapsayıcıdır. Extent içerisindeki page'ler veri satırlarını içeren kapsayıcılardır. Her extent'in içeriği page sayısı sekiz olsa da, her page içerisindeki satır sayısı aynı değildir. Page'ler veri satırlarını kapsayan en alt kapsayıcı olduğu için, veri büyülüğüne göre değişen satırlara sahiptir. Bir satır, sadece bir page içerisinde olabilir.

Extent ve page'ler ile ilgili açıklamaları daha iyi anlayabilmek için, yukarıdaki extent grafiğini inceleyerek, içeriği 8 ayrı page'i inceleyebilirsiniz. Her page içerisindeki yöntemi anlamak için de soldaki page grafiğini inceleyebilirsiniz.



İç içe hiyerarşi halinde çalışan bu yapılar, birbirini tamamlayarak SQL Server veritabanı ve içeriği nesnelerin yönetilmesini sağlar.

Page'ler veriler ile üst hiyerarşi arasında son kapsayıcı katmandır. Tüm veri ve nesneleri doğru ve performanslı yönetebilmek için kendi içerisinde farklı alt parçalara sahiptir.

Verilerin extent ve page'ler içerisindeki yönetimi, hangi page ve extent'lerde boş alan olduğu, bunların sıralaması, Indekslerin page'ler içerisinde saklanması, metin ve BLOB verilerin nasıl depolanacağı gibi kısımlar bu alt page'ler ile yönetilir.

Alt page'lere kısaca deşinerek hangi amaç ile kullanıldığını inceleyelim. Bu konular SQL Server mimarisi açısından önemli olsa da derin anlamda bilinmesi gereken konular değildir. Bu nedenle, özetleyerek inceleyeceğiz.

## **VERİ PAGE**

Verinin saklandığı veri page'dir. Veri page, verilerin tutulması için genel bir page olduğu için, BLOB veri tipindeki verileri de kapsaması gereklidir. BLOB veriler, BLOB page'de saklanır. Ancak BLOB page'deki adresi işareten eden 16 baytlık bir pointer da veri page'de saklanır. Bu sayede, veriler ile BLOB veri arasındaki bağı koparılmamış olur.

BLOB verileri varsayılan olarak büyük veri olarak nitelendirilir. Ancak, eğer metinsel veri page genişliğine siğacak şekilde kısa ise, BLOB veri ile birlikte saklanır. BLOB veri kısa değilse, BLOB page'de saklanır ve 16 baytlık bir pointer ile adresi de veri ile birlikte saklanır.

## **İNDEKS PAGE**

İndeks verisinin tutulduğu page'lerdir. Clustered Indeksin non-leaf page'lerini ve non-clustered Indeksin leaf ile non-leaf page'lerini içerisinde barındırır.

## **BLOB PAGE**

**BLOB** (*Binary Large Object*) verileri depolamak için kullanılır. BLOB veriler, çok geniş Text, Ntext veriler ile Image, Video gibi verilerin Binary olarak saklanması sağlar. Bu türden verileri işlemeyi ve yönetmeyi **İleri Sorulama Teknikleri** bölümünde detaylıca incelemiştik. BLOB veriler 2GB'a kadar geniş veriyi depolayabilecek kapasitededir. Page'lerin kapasitesini göz önünde

bulundurunca, bir page'e sıçma ihtimali yoktur. Bu nedenle, birden fazla page'de yer alabilecek verilerdir. Page yönetimi karmaşık olduğundan ve anlık olarak page sıralaması değiştireceğinin için, bu kadar çok page'in ardışık olarak yer olması düşünülemez. Bu nedenle, SQL Server BLOB verilerde, page'lerin ardışık olmasını garanti etmez.

## **GAM, SGAM, PFS**

Kısaltma olarak belirtilen page tiplerinin açılımı aşağıdaki gibidir.

**GAM** (*Global Allocation Map*)

**SGAM** (*Shared Global Allocation Map*)

**PFS** (*Page Free Space*)

Extent ve page'ler verileri tutan yapılardı. Page'lere sürekli veri yazma ihtiyacı olduğu için SQL Server, daha önceden gerektiği ayırdığı extent ve page'lerin boş ve dolu olma durumlarını kontrol eder. Bu kontrol işlemini gerçekleştirmek için de GAM, SGAM ve PFS page'lerini kullanır. Bu page'ler hangi extent ve page'lerin boş ve dolu olduğu bilgisini saklarlar. Bu sayede, SQL Server, hangi extent ve page'lere veri yazılabilceğini bilir.

## **BCM**

SQL Server'da dışarıdan veri alma ve tablonun Truncate edilmesi gibi işlemlere bulk işlemleri denir. Truncate ile ilgili bölümde anlattığımız gibi, bu işlem ile silinen veriler hızlı olarak page'lerden koparılırak silinir. Bu nedenle, log tutulmaz ve işlemler geri alınamaz. Log tutma işlemi sadece operasyondan kaynaklı temel olarak gerçekleştirilir. Ancak verileri geri alacak derecede özel bilgiler loglanmaz.

BCM (Bulk Changed Map), bulk işlemleriyle değiştirilmiş büyüklükleri izleyen sayfalar bütünüdür. BCM, değişikliklerin özellikle ilgilenemez ve veritabanı yedekleme işleminde daha fazla seçenek sunulması için kullanılır.

## **DCM**

**DCM** (*Differential Changed Map*)'de BCM gibi yedekleme işlemlerinde yardımcı rolünü oynar. BCM yedekleme işlemleri için kullanılırken, DCM'de alınan son yedekten sonraki değişiklikler için kullanılır.

## PAGE SPLIT

Page'in sınırlarından page konusunda bahsetmiştik. Page dolmaya başladığında, page bölünür. Bu işleme **Page Split** denir.

## SATIRLAR

Satır kavramı, veritabanı tablolarının her bir satırını ifade eder. Her satır maksimum 2014 sütundan oluşabilir. Yukarıda bahsettiğimiz alt yapı mimarisinden dolayı 8060 karakter sınırını geçemezler. Satırlar, bütün olarak tabloları oluşturur. Tablolar ve diğer nesneler ise veritabanını oluşturur.

## İNDEKSLER NERELEERDE KULLANILIR?

SQL Server'da Indekslerin performans için kullanıldığından bahsettik. Teorik anlamda öyledir. Ancak, nesnel olarak hangi durumlarda Indeksler kullanılır kısaca bunları inceleyelim.

- En basit tabir ile Indeksler, SQL Server'ın veri ve tablo yapısını doğru tanıması için kullanılır. SQL Server'ın doğru ve hızlı performans sergileyebilmesi için bu özellik bile yeterlidir.
- **Primary Key** ile noktasal sorguları hızlandırmak için kullanılabilir. Bu sorgu tipi, **WHERE** ile gerçekleşir ve tam olarak hangi kaydın istediği belirtildiği için, veriye en hızlı ulaşabilecek durumlardan biridir.

Örneğin;

---

```
SELECT Name FROM Production.Product WHERE ProductID = 1;
```

---

- Çok örneklediğimiz veri tekrarlarından kaçınmak için kullanılır. Bu bölümün Unique Indeks kısmında ve Constraint bölümünün Unique Constraint kısmında bu konuyu detaylarıyla inceledik.
- **ORDER BY** sıralama işlemlerinde, sıralamayı daha hızlı yapabilmek için kullanılır.
- Aralık sorgulama işlemlerinde kullanılır. Bu yöntem **WHERE** ile filtrelemeye benzer. Tek farkı, belirlenmiş aralıklardaki kayıtların getirilmesini sağlamaktır.

Bunlar ve daha birçok sebeple Indeksler kullanılabilir.

## İNDEKSLERİ ANLAMAK

İndeks mimarisini incelerken birçok alt yapı ve özellikten bahsettik. Indeksler, veri depolama mimarisinin veriler üzerindeki bir performans parçasıdır diyebiliriz. SQL Server ile verinin doğru yönetilmesi ve hızlı sorgulama gerçekleştirmek için kullanılırlar.

Bu bölüm ve sonrasında bölgümlerde, Indeksler hakkında teorik bilgilere az girerek, uygulama tarafında Indekslerin oluşturulması ve yönetilmesini inceleyeceğiz.

### CLUSTERED İNDEKS

Clustered Indeks'te tabloda yer alan kayıtlar, fizikal olarak Indeks tanımlı olan sütuna göre dizilirler. Clustered Indeks'lerin kaydedildiği sayfalar ile gerçek veri aynı seviyededir. Bu nedenle, doğru sütunlar üzerinde oluşturulan clustered Indeksler hızlı sonuç döndürürler.

İndeksler tanımlanmadan önce veritabanı yapısı ve kullanım yoğunluğu hesaplanarak iyi bir analiz süreci geçirilmelidir. Bir clustered Indeks oluşturmak için tablodaki en yoğun sorgu olması ön görülen ya da istatistikler ile en çok sorgu aldığı bilinen sütunlar seçilmelidir. **AdventureWorks** veritabanındaki **Product** tablosu üzerinde bir clustered Indeks tanımlamak için **ProductID** ya da **Name** sütunu kullanılabilir. Bu iki sütun da yoğun kullanılan sütunlardır. Ürünleri isimlerine göre aramanın yoğun olduğu bir mimari de **Name** sütunu üzerinde bir clustered Indeks oluşturmak yararlı olabilir. Ancak **AdventureWorks** veritabanı mimarisinde, en yoğun kullanılan sütun **ProductID** sütunudur. Bir tabloda sadece tek bir clustered Indeks oluşturulabilir. Bu nedenle, clustered Indeks oluştururken, sorgu ve tablo yapısı iyi analiz edilmeli ve seçici davranışılmalıdır.

Clustered Indeks olarak belirlenen sütunların tekil, yani benzersiz değerlere sahip olması gereklidir. Constraint işlemlerinden hatırlayacağınız üzere veri üzerinde tekilleştirmeyi sağlamak için **Unique Constraint** ya da **Primary Key Constraint** kullanılabilir. Bu constraint'ler ile veriyi tekilleştirme sağlanabilir. Performans ve Indeks mimarisi için verinin tekil olması gerekse de, genel kullanımda durum böyle değildir. Genelde tekrarlayan verilerin bulunduğu sütunlar üzerinde, veriye erişimi hızlandırmak için clustered Indeks oluşturulur. SQL Server, bu durumu arka planda çözmektedir. SQL Server, clustered Indeks oluşturulan sütun için 4Byte'lık tekilleştirici (*identifier*) kullanır. Bu sayede,

sütun içerisinde verinin tekilliği sağlanmamış görünse bile, arka planda SQL Server için veri tekil yani benzersizdir.

SQL Server, Indeks ihtiyacını aslında kendisi belirler. Programcı öneri olarak bazı Indeksler oluşturur. Ancak SQL Server, hangi Indeksin, hangi durumlarda ve ne şekilde kullanacağına kendisi karar verir.

## **CLUSTERED İNDEKS TARAMASI (SCAN)**

Sorgularda herhangi bir koşul yoksa kullanılır. Koşul ve sıralama işlemlerinin olmadığı durumlarda, Indeksler bakılmaksızın tüm tablo içeriği taranarak sonuç döndürülür. Bu yönteme **Table Scan** denir. Indeks kullanılmayan tablolarda da bu tarama yöntemi kullanılır.

## **CLUSTERED İNDEKS ARAMASI (SEEK)**

Sorgularda **WHERE** gibi bir koşul varsa kullanılır. Amacı, Indeksler üzerinden koşul ile belirtilen kayıtların bulunması ve sorgu içerisinde kullanılacak şekilde verilerin getirilmesini sağlamaktır.

## **NON-CLUSTERED İNDEKS**

Non-Clustered Indeksler, Clustered Indeksler gibi fiziksel değil, mantıksal olarak dizme işlemi gerçekleştirirler. Non-Clustered Indeksler, Clustered Indekslerin yardımcılarıdır diyebiliriz. Bir tablo üzerinde sadece bir clustered Indeks tanımlanabilirken, non-clustered Indeksten 999 adet tanımlanabilir. Bir Non-Clustered Indeks verilere doğrudan erişemez. Ancak, Heap üzerinden ya da bir Clustered Indeks üzerinden verilere erişebilir.

# **SQL SERVER İNDEKS TÜRLERİ**

SQL Server'da Indeksler, kullanım alanlarına ve teknik özelliklerine göre farklı türlere ayrırlar. Bu Indeks türleri aşağıdaki gibidir.

## **UNIQUE İNDEKS**

Verinin tekilliğini garanti etmek için kullanılır. Örneğin; email ile üyelik gerektiren durumlarda email adresinin benzersiz olması gereklidir. Bu senaryoda, tablodaki email sütununu Unique Indeks olarak tanımlamak sütundaki email bilgisinin benzersiz olmasını garanti eder.

Unique Indeks, hem veri tekrarını engeller hem de veri çekme hızını artırır.

## SÜTUNA KAYITLI (COLUMNSTORE) İNDEKS

Columnstore Indeks'ler, her bir sütuna ait verileri aynı sayfaya devam ettirerek sık kullanılan sorgularda performans artışı sağlar. Sık kullanılan sorgularda hızlı okuma gerçekleştirir.

SQL Server 2012 ile birlikte gelen bu yeni Indeks'in veriler üzerinden bir kısıtlaması vardır. Columnstore Indeks tanımlanmış bir tabloda sadece okuma yapılabilir. Tablodaki veri üzerinde ekleme, güncelleme ve silme işlemi gerçekleştirilebilir için Indeks pasifleştirilmelidir (*disable*).

## PARÇALI İNDEKS

İndeksleri farklı fiziksel dosya gruplarına dağıtarak sorgu performansını artırmak için kullanılır. SQL Server 2005 ile gelen bu Indeks türü; Clustered ya da Non-Clustered olabilir.

## EKLENTİ SÜTUNLU İNDEKS

İndeks yapısının en uç sayfalarında gerçek verilerde tutarak sorgu performansını hızlandırmak için kullanılan bu Indeks türü SQL Server 2005 ile birlikte gelmiştir.

Eski tip veri tipleri olan **TEXT**, **NTEXT** ve **IMAGE** türünden sütunlar eklenti sütun olarak kullanılamazlar. Ayrıca, bir eklenti sütunlu Indeks'in boyutu her satır için 900 baytı geçemez.

## XML İNDEKS

XML veriler ile native olarak tam uyumlu olan SQL Server, XML sütunlar için de sorgu performansını artırmak için Indeks oluşturmaya destek verir. XML'in yapısı gereği Indeks kullanımı da biraz farklıdır.

XML Indeks konusu XML bölümünde detaylıca anlatılmaktadır.

## KARMA (COMPOSITE) İNDEKS

16 sütun ya da toplam uzunlukları 900 baytı geçmemek üzere, birden fazla alanı kapsayan Indekstir.

## KAPSAM (COVERING) İNDEKS

Bir sorgunun `WHERE` kısmını da dahil ederek, seçilen sütunlar ile birlikte bir Indeks olarak tanımlanmasına denir. Diğer Indeks yöntemlerine göre farklı bir amaca hizmet ettiği ve kapsadığı verinin karmaşıklığı nedeniyle performans olarak yavaş olsa da, aynı işlemin Kapsam Indeksi kullanılmamış haline göre daha performanslıdır.

## FİLTRELİ İNDEKS

Adından da anlaşılacağı gibi, bir sütundaki tüm kayıtları Indekslemek yerine, sadece belirlenen kurala uya satırları Indekslemek amacı ile kullanılır. Filtreli Indeksler, SQL Server 2008 ile birlikte gelmiştir.

## FULL-TEXT İNDEKS

Full-Text Search özelliği için tasarlanan bu Indeks türü sadece `char`, `nchar`, `varchar`, `nvarchar(max)`, `varbinary(max)`, `image` ve `xml` veri tipindeki sütunlar üzerinde tanımlanabilir. Full-Text Indeksler kavramını anlayabilmek için Full-Text Search özelliğini bilmek gereklidir.

Özetlemek gerekirse, genellikle arama motoru amacı ile kullanılırlar. Doküman gibi yoğun içeriğe sahip veriler üzerinde dil kurallarından bağımsız hızlı sorgulamalar yapmak için kullanılır.

## İNDEKS OLUŞTURMAK

Buraya kadar, SQL Server depolama ve Indeks mimarisini inceledik. Artık bir Indeks oluşturabilmek için gereken temel bilgilere sahibiz. Indeks oluşturmak için gerekli söz dizimi en temel anlamda basittir. Ancak, Indeks kavramı çok geniş ve bir çok farklı özelliklere sahiptir. Bu nedenle, detaylı söz dizimi biraz karmaşık gelebilir.

Öncelikle basit Indeks oluşturma söz dizimini inceleyeceğiz. Sonrasında detaylı söz dizimi özelliklerini sırasıyla inceleyeceğiz.

### Söz Dizimi:

---

```
CREATE Indeks_tipi INDEX Indeks_ismi
ON tablo_ismi(sutun_ismi)
```

---

Basit söz diziminde her şey açıktır. Nesne oluşturmak için kullanılan **CREATE** ile başlayarak, önce Indeks tipi belirleniyor, daha sonra da bu Indekse bir isim veriyoruz. Bir sonraki satırda ise bu Indeksin hangi tablo üzerindeki hangi sütun için oluşturulacağını belirtiyoruz.

## İNDEKS OLUŞTURURKEN

### KULLANILAN İFADELER

- **indeks\_tipi:** CLUSTERED, NONCLUSTERED ya da UNIQUE CLUSTERED olarak belirtilir. Tip belirtilmezse varsayılan olarak NONCLUSTERED kullanılır.
- **indeks\_ismi:** Indekse verilen isimdir. Indeks tiplerinin baş harflerine göre kısa isimler alabilir (CL gibi).
- **tablo\_ismi:** Indeksin üzerinde tanımlandığı tablo ya da view'in ismi.
- **sutun\_ismi:** Tablo ya da view'de Indekslenmesi istenen sütun ya da sütunların isimleri.

## AYRINTILI İNDEKS SÖZ DİZİMİNİ ANLAMAK

İndeksler temel anlamda basit olduğu gibi birçok ileri seviye özelliğe sahiptir. Bu özelliklerin bazılarını az kullanabileceğiniz gibi bazılarını muhtemelen hiç kullanmayacaksınız. Ancak, Indekslerin güç ve yeteneklerinin farkında olabilmek için teorik anlamda öğrenilmesinde yarar vardır.

İndekslere ait özellikleri sırasıyla inceleyelim.

## ASC/DESC

İndeksin artarak ya da azalarak sıralanmasını belirtmek için kullanılır. **ASC** (*Ascending*) diğer SQL işlemlerinde olduğu gibi artan şekilde sıralama yapar ve varsayılan özelliktir. **DESC** (*Descending*) ise tersi yönde sıralama yaparak azalan şekilde sıralar.

Bir veriyi artan, diğer veriyi azalan şekilde sıralamak mümkündür. Bu durumda bir veri fiziksel olarak artan şekilde sıralanırken diğer veri fiziksel olarak azalan şekilde sıralanacaktır.

### Söz Dizimi:

---

[ ASC | DESC ]

---

## INCLUDE

İndeks türlerini anlatırken Kapsam Indekslerinden bahsetmiştir. **INCLUDE**, **Kapsam (Covering)** Indeksleri desteklemek amacı ile SQL Server 2005'de söz dizimine eklenmiştir. Bu yöntemde, gerekli veriler zaten Indekste yer aldığı için gerçek veri page'lerine tekrar erişmeye gerek yoktur. Gerçek veri page'lere gidilmeye gerek olmaması I/O performansı açısından faydalı bir özelliktir.

### Söz Dizimi:

---

```
INCLUDE (column [ ,... n ] )
```

---

## WITH

Kendisinden sonra gelen özelliklerin kullanılmasını sağlar.

## PAD\_INDEX

İndeks oluşturulduğunda non-leaf seviye page'lerin yüzdesel olarak nasıl dolu olarak kabul edileceğini belirler.

### Söz Dizimi:

---

```
PAD_INDEX = { ON | OFF }
```

---

## FILLCFACTOR

Page'lerin yoğunluğunu ayarlamak için kullanılır. Indeks ilk oluşturulurken, page'ler varsayılan olarak tam doluluk durumundan iki kayıt eksik olarak doldurulur.

Yüzdesel olarak, page'in dolu kabul edilmesi gereken oran belirtilebilir. Bunun için **FILLCFACTOR** değerini 1-100 arasında bir değer verilir. Bu değer sadece Indeks oluşturulurken değiştirilebilir. Mevcut bir Indeksin **FILLCFACTOR** değeri üzerinde değişiklik yapılamaz.

### Söz Dizimi:

---

```
FILLCFACTOR = fillfactor
```

---

## **IGNORE\_DUP\_KEY**

Unique Indeks ile veri tekrarını önlendiğinde, veri ekleme işlemlerinde veri tekrarını sağlayacak bir değer bulunduğuanda işlem hata verir. Bir transaction içerisinde bu işlemle karşılaşılırsa veri ekleme işlemi hata vereceği için transaction'da tamamlanamayacak ve sonlandırılacaktır.

Örneğin; transaction içerisindeki bir veri ekleme işleminde böyle bir hatanın olması sonucunda, transaction'ın sonlanması istenmeyebilir. **IGNORE\_DUP\_KEY** özelliği; hata mesajının seviyesini düşürerek bir uyarı mesajı halinde verilmesini sağlar. Böylelikle bir hata ile karşılaşılmadığı için transaction sonlanmayacak ve devam edecektir. Ancak, transaction sonlanmasa bile veri ekleme işlemi gerçekleştirilemeyecektir. Yani veri ekleme gerçekleşmez, ama transaction'da sonlandırılmaz.

### **Söz Dizimi:**

---

```
IGNORE_DUP_KEY = { ON | OFF }
```

---

## **DROP\_EXISTING**

Oluşturulmak istenen bir Indeks adı ile aynı isimde, yeni bir Indeks oluşturulmak istendiğinde eski Indeksi silip yeni Indeksi aynı isimle oluşturur. Gereksiz gibi görülebilir. Ancak, bazı durumlarda Indeksleri silmek hiç kolay olmayacağından emin olmak gereklidir. **DROP\_EXISTING** özelliği bu tür durumlarda performanslı bir şekilde eski Indeksi siler ve aynı isimle yeniden oluşturur.

### **Söz Dizimi:**

---

```
DROP_EXISTING = { ON | OFF }
```

---

## **STATISTIC\_NORECOMPUTE**

İndeksler için istatistikler hayatı öneme sahiptir. Bir Indeksin performanslı ve doğru yöntemle çalışabilmesi için Query Optimize, Indeksin istatistik bilgilerini kullanır. Bu bilgiler otomatik olarak SQL Server tarafından güncellenir. **STATISTIC\_NORECOMPUTE** özelliği kullanılarak bu otomatik gerçekleşen istatistik güncelleme işlemini kendinizin yapmak istediğiniz belirtmeniz anlamına gelir. Bu özellik kullanılırsa SQL Server istatistik güncelleme işlemini otomatik gerçekleştirmez ve geliştiricinin takip etmesini ve güncellemesini bekler.

Tablodaki veri ve tablo özellikleri değişikçe bu istatistiklerin güncellenmesi gereklidir. El ile istatistik güncellemek için **UPDATE STATISTICS** komutu çalıştırılmalıdır. Indeksler ile ilgili istatistik konusunu bölümün ilerleyen kısımlarında detaylıca inceleyeceğiz. Ancak, güncelleme işlemini SQL Server'a bırakmanız önerilir.

#### **Söz Dizimi:**

---

```
STATISTICS_NORECOMPUTE = { ON | OFF }
```

---

## **SORT\_IN\_TEMPDB**

Tempdb ve disk okuma-yazma işlemleriyle ilgili bir özelliktir. Indeksleri barındıran Tempdb, veritabanının bulunduğu fiziksel sürücüden farklı bir yerde depolandığı durumlarda bu özellik kullanılır.

**SORT\_IN\_TEMPDB** özelliği veritabanı yöneticiliği ile ilgili bir konu olduğu için bu detayları bu kitabın konusu dışındadır.

#### **Söz Dizimi:**

---

```
SORT_IN_TEMPDB = { ON | OFF }
```

---

## **ONLINE**

**ONLINE** özelliği, tabloyu erişime açmak olarak özetlenebilir. Kullanıcıların Indeks ya da tabloya erişimini engelleyen herhangi bir Indeks oluşturulamamasını sağlar.

Bu özellik, sadece SQL Server Enterprise Edition tarafından etkin olarak kullanılabilir. Daha küçük SQL Server versiyonlarında **ONLINE** özelliğini kullanılabilsse de herhangi bir etkisi olmayacağından emin olun.

#### **Söz Dizimi:**

---

```
ONLINE = { ON | OFF }
```

---

## ALLOW PAGE/PAGE LOCKS

İndeksin kilit biçimlerine izin verip vermeyeceğini belirler. Kilitler ile ilgili bölümde anlatılan tüm konuları incelediğinizde bu özelliğin ne kadar ileri seviye ve ustalık isteyen bir özellik olduğunu anlayabilirsiniz. Bu tür ileri seviye özellikleri kullanmak için veritabanı programlama ve yönetim alanlarında ileri seviye bilgi ve tecrübe sahib olmanız gereklidir. Aksi halde, içinden çıkışsız bir hal alabilir.

### Söz Dizimi:

---

```
ALLOW_ROW_LOCKS = { ON | OFF }
```

---

## MAXDOP

SQL Server'da her bir işlem için kullanılan ve işlemler için kaç işlemcinin kullanılacağını belirleyen **maksimum paralellik derecesi** adında bir sistem ayarı vardır. **MAXDOP** özelliği ile paralellik derecesi ayarlanabilir.

### Söz Dizimi:

---

```
MAXDOP = paralellik_olcusu
```

---

## ON

Performans ile ilgili ve ileri seviye bir SQL Server konusudur. Indekslerin verinin bulunduğu diskten farklı bir diskte saklanması için kullanılır.

## İNDEKSLER HAKKINDA BİLGİ EDİNMEK

Tablo ve view üzerindeki Indeksleri takip etmek önemlidir. Oluşturulan Indekslerin incelenmesi için SQL Server bir sistem prosedürü kullanır.

### Söz Dizimi:

---

```
sp_helpindex 'tablo_yada_view_ismi'
```

---

**Production.Product** tablosu üzerindeki Indeksleri inceleyerek bilgi edinelim.

```
EXEC sp_helpindex 'Production.Product';
```

	index_name	index_description	index_keys
1	AK_Product_Name	nonclustered, unique located on PRIMARY	Name
2	AK_Product_ProductNumber	nonclustered, unique located on PRIMARY	ProductNumber
3	AK_Product_rowguid	nonclustered, unique located on PRIMARY	rowguid

Sorgu sonucunda gelen sütunlar:

- **index\_name**: Indeksin adı.
- **index\_description**: Indeksin yapısı hakkında oluşturulan açıklama.
- **index\_keys**: Indeksin hangi sütunlar üzerinde oluşturulduğu. Indeks keyleri.

İndeksler hakkında bilgi almak için **sys.indexes** sistem kataloğu da kullanılabilir.

```
SELECT * FROM sys.indexes;
```

object_id	name	index_id	type	type_desc	is_unique	data_space_id	ignore_dup_key	is_primary_key	is_unique_constraint	fill_factor	is_padded	is_disabled	is_hypothetical
1	c1st	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
2	clust	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
3	c1st	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
4	clust	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
5	nc	2	2	NONCLUSTERED	1	1	0	0	0	0	0	0	0
6	NULL	0	0	HEAP	0	1	0	0	0	0	0	0	0
7	c1st	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
8	cl	1	1	CLUSTERED	1	1	0	0	0	0	0	0	0
9	nc	2	2	NONCLUSTERED	1	1	0	0	0	0	0	0	0
10	nc2	3	2	NONCLUSTERED	1	1	0	0	0	0	0	0	0

**sys.indexes** ile Indeksler hakkında tüm detaylı bilgi alınabilir.

İndekslerin ayrıntılı söz dizimini inceleyerek güç ve yeteneklerini öğrendik. Indeksler hakkında bilgi edinmeyi ve incelemeyi öğrendik. Şimdi, en temel Indeks söz dizimini kullanarak bir kaç örnek yaparak inceleyelim.

İndeks örneğinde kullanılacak **Personeller** tablosunun yapısı aşağıdaki gibidir.

□	dbo.Personeller
□	Columns
□	PersonelID (int, not null)
□	KullaniciAd (varchar(20), not null)
□	Email (varchar(50), null)
□	Sehir (varchar(30), null)
□	KayitTarih (smalldatetime, not null)

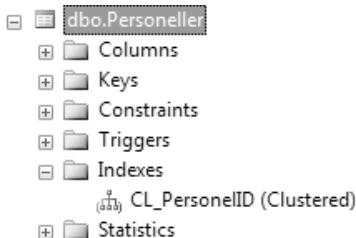
**Personeller** tablosunun **PersonelID** sütunu üzerinde **Clustered** Indeks tanımlayalım.

---

```
CREATE CLUSTERED INDEX CL_PersonelID
ON Personeller(PersonelID);
```

---

Indeks oluşturulduktan sonra, **Personeller** tablosu içerisinde **Indexes** kısmında görülebilir.



Yukarıdaki Indeks, varsayılan Indeks oluşturma yöntemi ile şu şekilde oluşturulabilirdi.

---

```
CREATE INDEX NC_PersonelID
ON Personeller(PersonelID);
```

---

Indeks söz diziminde Indeks tipini açıklarken eğer tip belirtilmezse, varsayılan olarak **NONCLUSTERED** Indeks oluşturulacağı belirtilmiştir. **CREATE INDEX** kullanımı varsayılandır. Indeks ismine de dikkat edilirse **NC** yani **NONCLUSTERED**'ın baş harflerinden oluşmaktadır.

İndeks söz dizimi ayrıntılarını açıklarken, ilk sırada açıklanan **ASC/DESC** ile sıralı Indeks oluşturma yöntemine de bir örnek verelim.

---

```
CREATE INDEX NC_PersonelID
ON Personeller(PersonelID ASC);
```

---

## UNIQUE İNDEKS OLUŞTURMAK

İndeksteki verilerin tekrarını önlemek için kullanılır ve **UNIQUE** deyimi ile tanımlanır. **UNIQUE** Indeks clustered ya da nonclustered olabilir. Bir Primary Key Constraint ya da Unique Constraint oluşturduğu zaman, SQL Server ilgili sütun için bir Unique Indeks tanımlar.

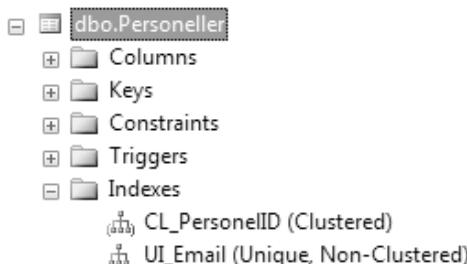
Personellerin email bilgilerini tutuyoruz. Her personel için benzersiz bir email adresi olması gereklidir. Veri girişi sırasında kullanıcıdan benzersiz bir email adresi girmesini istiyoruz. Bu nedenle `Email` sütunu üzerinde bir Unique Indeks oluşturalım.

---

```
CREATE UNIQUE NONCLUSTERED INDEX UI_Email
ON dbo.Personeller(Email)
ON [PRIMARY];
```

---

`UI_Email` isimli Indeks oluşturulduktan sonra tablodaki `Indexes` kısmının görünümü aşağıdaki gibidir.



Unique Constraint ile Unique Indeks oluşturulabildiği gibi, Primary Key Constraint ile de Unique Indeks oluşturulabilir. Primary Key Constraint kullanarak bir Unique Indeks oluşturalım.

## KAPSAM (COVERING) INDEKS OLUŞTURMAK

Karma Indeks ile ilgili açıklamayı, Indeks Türleri kısmında yapmıştık. Anlaşılmazı zor olabilecek bir Indeks olduğu için, örnek senaryo üzerinde inceleyelim.

Kitaptaki sorguları genel olarak `Production.Product` tablosu ile gerçekleştirdik. Ayrıca, tüm sütunlarını almak yerine 3 ya da 4 sütunu ile işlem yaptık. Soru sayısı yüksek olduğuna göre, gerçek uygulamalarda bu sorguların performansını artırmak için bir Indeks tanımlamak gereklidir. Birden fazla sütundan oluşan bir sorgu olduğuna göre en uygun Indeks tipi **Kapsam Indeks (Covering Index)** olabilir.

Sorgumuz aşağıdaki gibidir:

---

```
SELECT Name, ProductNumber, ListPrice FROM Production.Product;
```

---

Kapsam Indeksini tanımlayalım.

---

```
CREATE INDEX CV_Product
ON Production.Product(Name, ProductNumber, ListPrice);
```

---

## EKLENTİ SÜTUNLU İNDEKS OLUŞTURMAK

İndekse, bütün sütunları anahtar olarak vermek yerine, sadece arama kriterlerini anahtar olarak verip, geri kalan sütunları eklenti sütun olarak belirleyecek bir Indeks oluşturulabilir.

---

```
CREATE INDEX CV_SalesDetail
ON Sales.SalesOrderDetail(SalesOrderID)
INCLUDE(OrderQty, ProductID, UnitPrice)
```

---

## FİLTRELİ İNDEKS OLUŞTURMAK

SQL Server'da tablonun sadece belirli şartlara uygun satırları üzerinde Indeks tanımlayarak, Indeks boyutunu azaltmak ve sorgu süresini kısaltarak performansı artırmak mümkündür.

Aşağıdaki sorgu ile alınacak filtreli veri için bir Indeks oluşturacağız.

---

```
SELECT ProductID, Name, Color FROM Production.Product
WHERE Color IS NOT NULL
```

---

Seçili filtreli veriyi Indekslemek için aşağıdaki gibi bir Indeks oluşturalım.

---

```
CREATE INDEX FI_Product
ON Production.Product(ProductID, Name)
WHERE Color IS NOT NULL;
```

---

Filtreli Indeksler, sadece **nonclustered** türünden Indeksler için geçerlidir.

## İNDEKS YÖNETİMİ

Doğru Indeks oluşturmak zaman ve tecrübe gerektirir. Bir o kadar, oluşturulan bu Indeksleri yönetmek, istatistikler, Indekslerin yeniden derlenmesi, kapatılması ya da açılması, seçeneklerinin değiştirilmesi de bir o kadar zaman ve tecrübe gerektiren uzmanlıklardır.

Bu bölümde, tüm bu konuları inceleyeceğiz.

## İNDEKSLER ÜSTÜNDE DEĞİŞİKLİK YAPMAK

İndeksler oluşturulduktan sonra veri ve tablo yapısı sürekli değiştiği için belli aralıklarla performans iyileştirilmeleri yapılmalıdır. Bu işlemlerin gerçekleşmesi için kullanılan bazı yöntemler vardır. Bir Indeksin yeniden derlenmesi ya da seçeneklerinin değiştirilmesi gibi işlemler için Indeksler üzerinde değişiklikler yapılmalıdır.

### **REBUILD: İNDEKSLERİ YENİDEN DERLEMEK**

Bir Indeksi silip yenisini oluşturarak kapladığı alanı azaltmak ve yeniden yapılandırmak için kullanılır.

#### **Söz Dizimi:**

---

```
ALTER INDEX ALL ON tablo_ismi
REBUILD(secenekler)
```

---

**Production.Product** tablosundaki Indeksi **REBUILD** ile yeniden derleyerek **FILLFACTOR** özelliğini düzenleyelim.

---

```
ALTER INDEX ALL ON Production.Product
REBUILD WITH(FILLFACTOR = 90)
```

---

Aşağıdaki gibi aynı anda birden fazla özellikte düzenleme işleminde seçenek olarak kullanılabilir.

---

```
REBUILD WITH(FILLFACTOR = 90, SORT_IN_TEMPDB = ON)
```

---

**REBUILD** işleminde kullanılabilecek seçenekler aşağıdaki gibidir:

- **FILLFACTOR**
- **SORT\_IN\_TEMPDB**
- **IGNORE\_DUP\_KEY**
- **STATISTICS\_NORECOMPUTE**

## REORGANIZE: İNDEKSLERİ YENİDEN DÜZENLEMEK

Bosalan Indeks sayfalarının atılmasını sağlar. Bu işlem Indeks performansı açısından yararlıdır. **REORGANIZE** deyimi sadece bu işlem için kullanıldığından, sadece kendisiyle ilgili olan **LOB\_COMPACTION** seçeneğini kullanabilir.

Unique Indeks olarak oluşturulan **UI\_Email** Indeksini yeniden derleyelim.

---

```
ALTER INDEX UI_Email
ON dbo.Personeller
REORGANIZE WITH (LOB_COMPACTION = ON);
```

---

## İNDEKSLERİ KAPATMAK

Çok sık rastlanan durum olmasa da, bazen tablolar üzerindeki Indeksleri bir süreliğine kullanıma kapatmak gerekebilir. Kritik bir konudur. Çünkü tablo üzerindeki Clustered Indeks kapatılırsa, tabloda en basit veri seçme sorusu dahi kullanılamaz. Nonclustered bir Indeks kapatılırsa veri seçme açısından sorun teşkil etmeyecektir.

**DIJİBİL** veritabanında daha önce oluşturduğumuz **Personeller** tablosunda bir Clustered Indeks tanımlamıştık. Bu Indeksi kapatalım.

---

```
ALTER INDEX CL_PersonelID
ON Personeller
DISABLE
```

---

 **CL\_PersonelID (Clustered)**

CL ile tanımladığımız bir Indeks, yani Clustered Indeks. Bu durumda veri seçme işlemi gerçekleşmemesi gerekiyor. Şimdi **Personeller** tablosunda veri seçme işlemi yapalım.

---

```
SELECT * FROM Personeller;
```

---

Msg 8655, Level 16, State 1, Line 1

The query processor is unable to produce a plan because the index 'CL\_PersonelID' on table or view 'Personeller' is disabled.

Clustered Indeksin kapatılması sonucunda artık veri seçme işlemi için bile bir sorgu çalıştırılamıyor.

Nonclustered, bir Indeksi kapatırken durum bu kadar katı değildir. **Production.Product** tablosu üzerinde oluşturulan **FI\_Product** isimli Nonclustered Indeksi kapatalım.

---

```
ALTER INDEX FI_Product
ON Production.Product
DISABLE
```

---

#### FI\_Product (Non-Unique, Non-Clustered, Filtered)

Kapatılan Indeksin bulunduğu tabloda veri çekme sorgusu gerçekleştirildiğinde herhangi bir hata vermeden sorgu sonucunu getirdiği görülebilir.

---

```
SELECT * FROM Production.Product;
```

---

Kapatılan Indeksleri açmanız önemlidir. Örneğin; Clustered Indeksi açmazsanız tablo üzerinde işlem yapamazsınız. **Personeller** tablosundaki Clustered Indeks ve **Production.Product** içerisindeki **FI\_Product** Indeksinin açılmasını isteyebiliriz.

İki şekilde Indeks açılabilir. Bunlar;

---

```
ALTER INDEX CL_PersonelID
ON dbo.Personeller
REBUILD

ALTER INDEX FI_Product
ON Production.Product
REBUILD
```

---

## İNDEKS SEÇENEKLERİNİ DEĞİŞİSTİRMEK

İndeks dizimindeki seçenekleri anlatırken birçok farklı özellikten bahsettim. Bu seçeneklerin ayarları değiştirilebilmektedir.

**FI\_Product** Indeksinin **ALLOW\_ROW\_LOCKS** özelliğini değiştirmeden önce **sys.indexes** içerisinde **ALLOW\_ROW\_LOCKS** sütununun değerine bakalım.

---

```
SELECT Object_ID, Name, Index_ID, Type, type_desc, Allow_Row_Locks
FROM sys.indexes WHERE Name = 'FI_Product';
```

---

	Object_ID	Name	Index_ID	Type	type_desc	Allow_Row_Locks
1	1973582069	FI_Product	12	2	NONCLUSTERED	0

**FI\_Product** Indeksinin **ALLOW\_ROW\_LOCKS** özelliğini değiştirelim.

---

```
ALTER INDEX FI_Product
ON Production.Product
SET (ALLOW_ROW_LOCKS = ON);
```

---

**sys.indexes** içerisindeki **FI\_Product** Indeksinin **ALLOW\_ROW\_LOCKS** özelliği değerine tekrar bakalım.

Object_ID	Name	Index_ID	Type	type_desc	Allow_Row_Locks
1	FI_Product	12	2	NONCLUSTERED	1

Sisteminizde **ALLOW\_ROW\_LOCKS** değeri daha önceden **ON** olarak ayarlı ise, aradaki farkı göremeyebilirsiniz.

Farkı görmek için **SET(ALLOW\_ROW\_LOCKS = OFF)** ile özelliği kapatarak tekrar deneyebilirsiniz.

Bir Indeksin aşağıdaki seçenekleri değiştirilebilir.

- **ALLOW\_PAGE\_LOCKS**
- **ALLOW\_ROW\_LOCKS**
- **STATISTICS\_NORECOMPUTE**
- **IGNORE\_DUP\_KEY**

## İSTATİSTİKLER

SQL Server, sorgu performansı için istatistikler tutar. Bu istatistikleri SQL Server tutmamasını ve el ile takip etmek de sağlanabilir.

Bunun için Indeks özelliklerinden **STATISTICS\_NORECOMPUTE** özelliğini tekrar inceleyebilirsiniz.

İstatistikler diğer veritabanı nesneleri gibi **CREATE** ile oluşturulur. İstatistikler güncellenebilir ve silinebilirler.

## İSTATİSTİK OLUŞTURMAK

İndekslerde olduğu gibi Indekslerde olmayan bir sütun için bile istatistik oluşturulabilir.

**Söz Dizimi:**

---

```
CREATE STATISTICS istatistik_ismi  
ON {tablo_ismi | view_ismi}(sutun_ismi)
```

---

**Production.Product** tablosundaki **ProductID** sütunu için bir istatistik oluşturalım.

---

```
CREATE STATISTICS Statistic_ProductID  
ON Production.Product(ProductID);
```

---

## **İSTATİSTİKLERİ SİLMEK**

Kullanıcı tarafından oluşturulan istatistikler silinebilirler.

**Söz Dizimi:**

---

```
DROP STATISTICS {tablo_ismi | view_ismi}.(istatistik_ismi)
```

---

İstatistik silebilmek için, istatistik nesnesinin bağlı olduğu tablo ya da view adı ile birlikte belirtilmesi gereklidir.

---

```
DROP STATISTICS Production.Product.Statistic_ProductID;
```

---

# SCRIPT VE BATCH KULLANIMI

SQL Server'da şu ana kadar birçok script hazırladık. Bu script'ler ile nesne oluşturduk (**CREATE**), nesne düzenledik (**ALTER**), veri çektiğimiz (**SELECT**). Daha bir çok işlemi, bu script'leri kullanarak gerçekleştirdik.

Bu bölümde, script ve batch kavramları ile **SQLCMD** komut satırı aracını inceleyeceğiz.

## SCRIPT TEMELLERİ

Script kavramı birçok yazılım geliştirme teknolojisinde kullanılan bir terim olmakla birlikte teknik olarak bir dosyaya kaydedilmiş olmayı gerektirir. SQL script'leri metin dosyaları olarak, **.sql** dosya uzantısı ile saklanır.

Script'ler belli görevleri gerçekleştirmek için oluşturulur. Bu nedenle genel olarak tek dosya halinde çalıştırılırlar. Bir script dosyasının içerisindeki kodlar ayrı ayrı çalıştırılmazlar. Script'ler sistem fonksiyonları ve lokal değişkenleri kullanabilirler.

Bir script yapısını kavramak için birçok özelliğin kullanıldığı bir örnek yapalım.

---

```
USE AdventureWorks
GO
DECLARE @Ident INT;
INSERT INTO Production.Location(Name, CostRate, Availability,
ModifiedDate)
VALUES ('Name_Degeri', 25, 1.7, GETDATE());
```

```
SELECT @Ident = @@IDENTITY;
SELECT 'Eklenen satır için identity değeri : '+CONVERT(VARCHAR(3),@Ident);
```

(No column name)
1 Eklenen satır için identity değeri : 61

Hazırladığımız bu script'i **USE**, **INSERT**, **SELECT**, **CONVERT**, lokal değişken ve sistem fonksiyonu gibi birçok ifade kullanılarak geliştirdik.

## USE İFADESİ

**USE** ifadesi geçerli olması istenen veritabanını seçer. SSMS'in **CTRL+U** kısayolu ile ulaşabildiğiniz Available Databases menüsünden de seçilebilir. SSMS'nin menü ile yaptığı bu işlemin T-SQL karşılığı olarak **USE** ifadesi kullanılır. **USE** ifadesi kullanılmadığında, script çalıştırıldığında geçerli olan veritabanında çalıştırılır.

**USE** ifadesini kullanmak zorunlu değildir. Ancak yoğun kod bloklarında hedefi tam belirleyerek herhangi bir yanlış işleme sebebiyet vermemek için kullanılması önerilir. Her veritabanı programcısının birçok kez yaptığı hatalardan biri, geçerli veritabanını belirlemeyi unutmak ve bu nedenle farklı veritabanlarında oluşturulan tablo gibi nesneler ya da daha büyük sorun yaşatabilecek işlemlerdir. Bu nedenle hazırladığınız script'lerin bir dosya olarak çalıştırılacağını unutmadan **USE** ifadesi kullanılıp kullanılmaması konusunda bir kez daha düşünmelisiniz.

## DEĞİŞKEN BİLDİRİMİ

Değişken bildirimini için **DECLARE** ifadesi kullanılır. Kullanımı basittir.

```
DECLARE @degisken_adi degisken_tipi,
        @degisken_adi degisken_tipi
```

Script temelleri kısmını anlatırken **@Ident** adında bir değişken tanımlamıştık. Şimdi basit bir değişken tanımı gerçekleştirelim.

```
DECLARE @sayi1 INT;
DECLARE @sayi2 INT;
DECLARE @toplam INT = @sayi1 + @sayi2;
SELECT @toplam;
```

(No column name)
1 NULL

Yukarıdaki işlemde `@sayi1`, `@sayi2` ve `@toplam` adında 3 değişken tanımladık.

---

```
DECLARE @sayi1 INT;
DECLARE @sayi2 INT;
```

---

Değişkenleri tanımlarken varsayılan değer atamak zorunda değiliz. Ancak bu durumda yukarıdaki script'i çalıştırıldığımızda `NULL` değerinin döndüğünü göreceksiniz. Çünkü değer ataması yapılmayan değişkenler `NULL`'dır.

Değişken tanımı sırasında değer atayarak tekrar çalışıralım.

---

```
DECLARE @sayi1 INT = 12;
DECLARE @sayi2 INT = 12;
DECLARE @toplam INT = @sayi1 + @sayi2;
SELECT @toplam;
```

	(No column name)
1	24

Değer atayarak çalıştığımız sorgu sonucunda dönen değer **24** olacaktır.

Değişken tanımlarken her değişken tanımını `DECLARE` ile ayrı ayrı yapmak genel kullanılan yöntemdir. Ancak tek satır ve tek `DECLARE` kullanarak yan yana, aralarına virgül koyarak da değişken tanımlanabilir.

Veritabanı programlamada değişkenler hayatı öneme sahiptir. Kullanımı sırasında hazırladığınız sorguda değişkenden gelen değerin sizin istediğiniz sonuç olmama durumunu hesap etmelisiniz. Integer bir değer beklediğiniz bir değişkenden ön göremediğiniz bir sebepten ya da hata yönetimini doğru gerçekleştiremediğiniz için dönecek bir `NULL` değeri, sorgunuzun hata vermesine sebep olacaktır.

## **SET İFADESİ KULLANILARAK DEĞİŞKENLERE DEĞER ATANMASI**

`SET` ifadesi, T-SQL'de tanımlanan değişkenlere değer atamak için kullanılır.

KDV oranı hesaplayan iki değişken oluşturalım. Bu örnekte KDV değerini nümerik olarak değişken kullanmadan veriyoruz.

---

```
SET @Fiyat = 10;
SET @KdvMiktari = @Fiyat * 0.18;
```

---

Aynı işlemi gerçekleştiren örneği KDV değerini değişken ile belirterek yapalım.

---

```
SET @Fiyat = 10;
SET @KdvOran = 0.18;
SET @KdvMiktar = @Fiyat * @KdvOran;
```

---

Şimdi bu işlemi gerçek örnek ile gerçekleştirelim. Örneğimizde değişkenleri tanımlayıp, gerçek değerler atayarak ekranda görüntüleyeceğiz.

---

```
DECLARE @Fiyat MONEY;
DECLARE @KdvOran MONEY;
DECLARE @KdvMiktar MONEY;
DECLARE @Toplam MONEY;
SET @Fiyat = 10;
SET @KdvOran = 0.18;
SET @KdvMiktar = @Fiyat * @KdvOran;
SET @Toplam = @Fiyat + @KdvMiktar;
SELECT @Toplam;
```

---

(No column name)	
1	11,80

**DECLARE** ile tanımladığımız değişkenleri tek satırda da tanımlanabilir.

---

```
DECLARE @Fiyat MONEY; @KdvOran MONEY, @KdvMiktar MONEY, @Toplam MONEY;
```

---

Değişkene değer atamayı veritabanındaki veriler üzerinde de gerçekleştirebiliriz.

**Production.Product** tablomuzdaki **ListPrice** sütunu ürünlerin fiyatını tutmaktadır. Bu fiyatlar arasında en yüksek değere sahip olanı değişkenler yardımı ile bulalım.

---

```
DECLARE @EnYuksekFiyat MONEY;
SET @EnYuksekFiyat = (SELECT MAX(ListPrice) FROM Production.Product);
SELECT @EnYuksekFiyat AS EnYuksekFiyat;
```

---

EnYuksekFiyat	
1	3578,27

Yukarıdaki işlemin sonucunun doğruluğu, şu şekilde test edilebilir.

---

```
SELECT ListPrice FROM Production.Product ORDER BY ListPrice DESC;
```

---

## **SELECT İFADESİ KULLANILARAK DEĞİŞKENLERE DEĞER ATANMASI**

**SELECT** ifadesi değişkenlerden gelen değerlerin atanmasında kullanılır. Bu ifadeyi bir önceki **SET** ifadesindeki örneklerle inceleyelim.

---

```
DECLARE @EnYuksekFiyat MONEY;
SELECT @EnYuksekFiyat = MAX(ListPrice) FROM Production.Product;
SELECT @EnYuksekFiyat AS EnYuksekFiyat;
```

---

EnYuksekFiyat	
1	3578,27

**SET** kullanımına göre **SELECT**'in daha sade, anlaşılır ve az kod ile geliştirildiğini fark etmiş olmalısınız. Ancak bunun böyle olması her zaman **SELECT** ile değişken ataması gerçekleştireceğiniz anlamına gelmez. İkisinin de gerekli olduğu farklı özellikleri ve kullanım alanları vardır.

### **SET ya da SELECT kullanımına nasıl karar vereceğiz?**

Örneklerden de anlayacağınız gibi **SET** kullanımında değişken değerlerinin belli olduğu örneklerde kullanırken, **SELECT** işleminde veritabanından veri çekerken kullandık. Bu her zaman böyle olmayacağından emin olmak gereklidir. Ancak genel kullanım açısından bu durum belirgin bir özelliklektir.

- Veritabanından sorgu sonucu elde edilen değişken değer atamaları için **SELECT** kullanın.
- Değişkenden değişkene ya da değeri bilinen bir değer ataması için **SET** kullanın.

## **BATCH'LER**

Batch, T-SQL ifadelerinin tek bir mantıksal birim içinde gruplandırılmasıdır. Batch içindeki tüm ifadeler bir uygulama planı içerisinde birleştirilir, bir arada derlenir ve söz diziminin doğruluğu onaylanır. Derleme sırasında hata meydana gelirse hiçbir ifade çalışmayaçaktır. Ancak çalışma zamanında meydana gelen hatadan önce çalışan ifadelerin sonuçları geçerlidir.

Onceki bölümde gördüğümüz script'lerin her biri bir batch oluşturur. Bir script'i birden fazla batch'e ayırmak için **GO** ifadesini kullanırız.

## BATCH'LERİ NE ZAMAN KULLANIRIZ?

Batch'ler, script'te bazı şeylerin script'teki diğer ifadelerden önce ya da farklı olarak uygulanması gerekiğinde kullanılır.

Batch içinde yer olması gereken ifadeler;

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

Batch yönetimini hatasız ve mantıksal olarak gerçekleştirebilmek için batch içerisindeki ifadeleri doğru kullanmalısınız. Bir batch'de **DROP** ile nesne silmek istiyorsanız, bu ifadeyi **CREATE** ifadesinden önce kullanmalısınız. **DROP** ifadesini ayrı olarak ve **CREATE** ifadesinden önceki bir önceki batch olarak çalıştırmanız gereklidir. Çünkü **DROP** ile sildiğiniz bir nesneyi **CREATE** ile tekrar oluşturmak istediğinizde, nesne isminin aynı olması durumunda olası bir hatadan kurtulmuş olursunuz.

## GO İFADESİ

**go** komutu, bir T-SQL komutu değildir. SQL Server'ın **SSMS** ve **SQLCMD** gibi geliştirme ve düzenleme araçları tarafından tanımlanmış bir komuttur. Bu araçlar haricindeki üçüncü-parti (*third-party*) araçlar **go** ifadesini desteklemeyebilir.

## GO İFADESİ NE İŞE YARAR?

Geliştirme araçlarında hazırlanan T-SQL kodlar veritabanı sunucusuna gönderilmek istendiğinde, **go** komutu yazılarak sorgunun çalıştırılmaya hazır olduğu ve işleme alınabileceğini belirtir.

Geliştirme aracı, **go** ifadesi ile karşılaştığında, batch'i sonlandırır ve paketleyerek tek bir birim olarak sunucuya gönderir. Ancak, sunucuya gönderilen T-SQL sorguları içerisinde **go** ifadesi yer almaz. Veritabanı sunucusu **go** ifadesinin ne anlama geldiğini dahi bilmez.

## S Q L C M D

SQLCMD, T-SQL script'lerini Windows komut ekranında çalıştırmak için kullanılan bir yardımcı SQL Server aracıdır. Toplu sorguların bulunduğu **.sql** dosyalarının çalıştırılması için iyi bir araçtır. Genel olarak bakım, yedekleme, veritabanı oluşturma ya da benzeri toplu sorguların bulunduğu **.sql** dosyalarının çalıştırılması için kullanıldığından dolayı, SQL Server veritabanı yöneticileri daha çok tercih etmektedir.

**SQLCMD**, eski versiyonu olan **OSQL**'in yerini alması için geliştirilmiştir.

### SQLCMD Söz Dizimi:

---

Sqlcmd

```
[ -U login id] [ -P password] [ -S server] [ -H hostname]
[ -E trusted connection] [ -d use database name] [ -l login timeout]
[ -N encrypt connection] [ -C trust the server certificate]
[ -t query timeout] [ -h headers] [ -s colseparator] [ -w screen width]
[ -a packetsize] [ -e echo input] [ -I Enable Quoted Identifiers]
[ -c cmdend] [ -L[c] list servers[clean output]] [ -q "cmdline query"]
[ -Q "cmdline query" and exit] [ -m errorlevel] [ -V severitylevel]
[ -W remove trailing spaces] [ -u unicode output]
[ -r[0|1] msgs to stderr] [ -i inputfile] [ -o outputfile]
[ -f <codepage> | i:<codepage>[,o:<codepage>]]
[ -k[1|2] remove[replace] control characters]
[ -y variable length type display width]
[ -Y fixed length type display width]
[ -p[1] print statistics[colon format]]
[ -R use client regional setting] [ -b On error batch abort]
[ -v var = "value" ...]
[ -X[1] disable commands[and exit with warning]]
[ -? show syntax summary]
```

---

SQLCMD sorgu ekranını açmak için;

- **Başlat -> Çalıştır -> cmd**
- **Başlat -> Çalıştır -> sqlcmd**

... yazarak giriş yapabileceğiniz gibi, sqlcmd sorgularınızı direkt olarak **Çalıştır** kısmında da yazabilirsiniz. **Direkt Çalıştır** kısmında tüm sorguyu yazacaksanız, sorgunun başında SQLCMD yazmış olmalısınız.

Komut ekranında `sqlcmd /?` yazarak komut satırının desteklediği SQLCMD sorgu parametrelerini görebilirsiniz.

```
C:\Users\dijibil>sqlcmd /?
Microsoft (R) SQL Server Command Line Tool
Version 11.0.2100.60 NT x64
Copyright (c) 2012 Microsoft. All rights reserved.

usage: Sqlcmd      [-U login id]          [-P password]
                   [-S server]           [-H hostname]         [-E trusted connection]
                   [-N Encrypt Connection] [-C Trust Server Certificate]
                   [-d use database name] [-l login timeout]   [-t query timeout]
                   [-h headers]          [-s colseparator]    [-w screen width]
                   [-a packetsize]        [-e echo input]     [-I Enable Quoted Identifiers]
                   [-c cmdend]           [-L[cl] list servers[clean output]]
                   [-q "cmdline query"]  [-Q "cmdline query" and exit]
                   [-m errorlevel]        [-V severitylevel]  [-W remove trailing spaces]
                   [-u unicode output]   [-r[0|1] msgs to stderr]
                   [-i inputfile]         [-o outputfile]     [-z new password]
                   [-f <codepage> | i:<codepage>[,o:<codepage>]] [-Z new password and exit]
                   [-k[1|2] remove[replace] control characters]
                   [-y variable length type display width]
                   [-Y fixed length type display width]
                   [-p[1] print statistics[:colon format]]
                   [-R use client regional setting]
                   [-K application intent]
                   [-M multisubnet failover]
                   [-b On error batch abort]
                   [-v var = "value"...] [-A dedicated admin connection]
                   [-X[i] disable commands, startup script, environment variables [and exit]]
                   [-x disable variable substitution]
                   [-? show syntax summary]

C:\Users\dijibil>
```

SQLCMD üzerinden sunucuya bağlanmak için bir çok farklı parametre kullanılabilir.

Bu parametrelerden bazıları;

- **s** : Sunucu adı
- **u** : Kullanıcı adı
- **p** : Kullanıcı parolası
- **H** : Host adı
- **E** : Güvenilir Bağlantı (*Trusted Connection*)

SQLCMD komut satırındaki parametrelerin küçük büyük harf duyarlı olduğunu bilmeniz gereklidir. Bu kısa isimlendirmeler haricinde de bir çok parametre kullanılabilir. Ancak temel olarak bunları kullanarak sunucuya bağlanabilirsiniz.

Bir bağlantı örneği;

---

```
sqlcmd -S DIJIBIL-PC
```

---

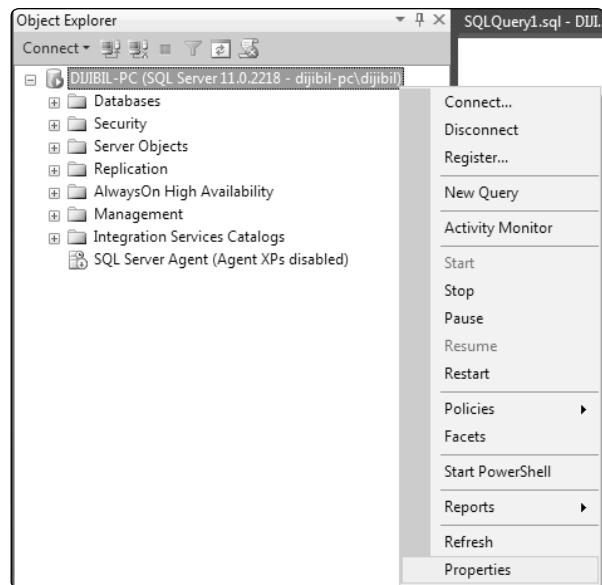
```
C:\Users\dijibil>sqlcmd -S DIJIBIL-PC
1>
```

Bu sorgu ile kullanıcı adı ve şifresi kullanmadan en basit şekilde bulunduğuuz bilgisayardaki SQL Server'a bağlanabilirsiniz.

Bilgisayarında sunucu adı olarak **DJIBIL-PC** kullanıldığı için sorguyu buna göre hazırladım. Siz, kendi bilgisayarınızdaki sunucu adını şu şekilde öğrenebilirsiniz.

- SSMS içerisinde Solution Explorer'da, soldaki ağaç yapısının en üstteki sunucu ismini kullanabilirsiniz.

Sunucu adı biraz uzun ise, hata yapmamak için, **Solution Explorer**'daki bu sunucu adına sağ tıklayarak **Properties** ekranında en yukarıda **Name** özelliğinden kopyalayabilirsiniz.



Name	DJI BIL-PC
Product	Microsoft SQL Server Enterprise Evaluation (64-bit)
Operating System	Microsoft Windows NT 6.1 (7601)
Platform	NT x64
Version	11.0.2218.0
Language	İngilizce (Amerikan)
Memory	8173 (MB)

Sunucuya kullanıcı adı ve şifresi ile bağlanmak için;

---

```
sqlcmd -Ssunucu_ismi123 -Uuser_name123 -Ppwd123
```

---

Bu şekilde, parametreler ile değerleri arasında boşluk olmayacak şekilde de SQ Server'a bağlanılabilir.

Sunucuya bağlandık. Şimdi ilk sorgularımızı yazarak sunucu hakkında bilgiler edinelim.

Sunucu adını listeleyelim.

---

SELECT @@SERVERNAME;	(No column name)
GO	1 DIJIBIL-PC

---

Sunucu versyonunu listeleyelim.

---

```
SELECT @@VERSION;
GO
```

---

(No column name)	1 Microsoft SQL Server 2012 - 11.0.2218.0 (X64) Jun 12 2012 13:05:25 Copyright (c) Microsoft Corporation Enterprise Evaluation Edition (64-bit) ...
------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------

Sunucu ve sorgularımız başarılı bir şekilde çalışıyor. Artık **AdventureWorks** veritabanımıza erişerek, istediğimiz sorguları çalıştırabilirim.

Öncelikle, SQLCMD içerisinde **AdventureWorks** isimli veritabanına bağlanarak oturum açmalıyız.

---

```
USE AdventureWorks
GO
```

---

Bu soruyu çalıştırıldığınızda ekranda şu mesaj belirecektir.

 Messages
 

Command(s) completed successfully.

Bu mesaj ile birlikte, artık ilgili veritabanına bağlanmış durumdasınız. Tüm sorgularınız, belirtilen veritabanı içerisinde çalıştırılacaktır.

Kendi bilgisayarımda, **AdventureWorks** veritabanının 2012 versiyonunu, **AdventureWorks2012** adı ile kullandığım için, bu ismi belirtmem gerekiyordu. Siz hangi veritabanı ile çalışıyorsanız o veritabanının ismini belirtmelisiniz.

Ürünler tablosundaki **ProductID** değeri 1 olan kaydı listeleyelim.

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID = 1;
```

	ProductID	Name
1	1	Adjustable Race

**SQLCMD** ekranından çıkarak, tekrar Windows komut satırı ekranına dönmek için, bu iki komut kullanılabilir.

- **quit**
- **exit**

## AKIŞ KONTROL İFADELERİ

Tüm programlama dillerinde bulunan akış kontrolleri, programsal akışı şartları sağlamaşı, akışın değiştirilmesi, belli görevin belli şartlarda gerçekleşmesi ya da belli görevlerin belirli defa gerçekleşmesi gibi işlemler için kullanılır. **Profesyonel** veritabanı uygulamalarında, akış kontrollerinin kullanılması kaçınılmaz gerekliliktir.

Akış kontrolleri ve yardımcı ifadeleri;

- **IF . . . ELSE**
- **CASE**
- **WHILE**
- **WAITFOR**
- **GOTO**

Programlama dillerinden herhangi birinde tecrübesi olan geliştiriciler bu akış kontrollerini belki yüzlerce ya da binlerce kez kullanmış olabilir. T-SQL'de de amaçları aynı olmakla birlikte sadece söz diziminde değişiklikler vardır.

## IF ... ELSE

Tüm programlama dillerinde var olan ve en çok kullanılan akış kontrol ifadelerinden biridir. Durum kontrolü için kullanılan **IF ... ELSE** akış kontrol yapısı tek bir şartı kontrol edebileceğ gibi birçok şartı da kontrol edebilir.

### Söz Dizimi:

---

```
IF <Boolean Deyimi>
    <SQL İfadesi> | BEGIN <kodlar> END
[ELSE
    <SQL İfadesi> | BEGIN <kodlar> END]
```

---

IF kullanımı için basit bir örnek yapalım.

---

```
DECLARE @val INT;
SELECT @val = COUNT(ProductID) FROM Production.Product;
IF @val IS NOT NULL
    PRINT 'Toplam ' + CAST(@val AS VARCHAR) + ' kayıt bulundu.';
```

---

Yukarıdaki sorguda, **@val** isimli **INT** veri tipinde bir değişken tanımladık. **SELECT** sorgusunda kullandığımız **COUNT** fonksiyonu ile ürünlerin sayısını toplayarak **@val** değişkenine aktarıyoruz. Eğer toplama işlemi sonucunda bir kayıt dönerse, kayıt toplamını ekranda göstermesini istiyoruz.

Bu işlemin gerçekleşebilmesi için bir akış kontrolü yapılması gereklidir. **IF** kontrol yapısını kullanarak, **@val** değişkeninin **NULL** olup olmadığını öğreniyoruz. Eğer **NULL** ise, bu sorgu sonucunda herhangi bir işlem yapılmayacaktır. Ancak **NULL** değil ise, **PRINT** komutu ile ekranda kaç kayıt bulunduğu göstererek.

**IF** kontrolünün en çok kullanıldığı durumlardan biri de, yeni bir tablo oluştururken, bu tablonun veritabanında var olup olmadığıdır. Eğer tablo var ise silinecek ve yenisini oluşturulacak, yok ise silme işlemine gerek kalmadan sadece **CREATE** sorgusu çalışacaktır.

Deneme isimli bir tablo veritabanında var ise sil ve yenisini oluştur. Bu isimde bir tablo yok ise, sadece oluştur.

---

```
IF EXISTS (
    SELECT *
    FROM Sys.Sysobjects WHERE ID = Object_ID(N'[dbo].[Deneme]')
    AND OBJECTPROPERTY(ID, N'IsUserTable') = 1
)
DROP TABLE dbo.Deneme;
CREATE TABLE dbo.Deneme
(
    DeneID INT
);
```

---

**IF** akış kontrolünde, **ELSE** kullanılarak akış kontrolünde belirtilen şart ya da şartlar sağlanmadığı takdirde çalışacak sorgular oluşturulabilir.

Ürünler tablosundaki kayıtları toplayalım. Eğer kayıt yok ise (yani **NULL** ise), ekranda 'Kayıt Yok' yazsın, eğer kayıt var ise, bu kayıtları toplayarak, toplam sonucunu veri tipi dönüştürme işleminden sonra bir metinsel değer ile birleştirilerek ekrana yazdıralım.

---

```
DECLARE @val INT;
SELECT @val = COUNT(ProductID) FROM Production.Product
IF @val IS NULL
    PRINT 'Kayıt Yok'
ELSE
    PRINT 'Toplam ' + CAST(@val AS VARCHAR) + ' kayıt bulundu.';
```

---



## ELSE Koşulu

Bazen bir ya da iki koşuldan daha fazlası için **IF ... ELSE** kullanılması gerekebilir. Bu tür durumlarda **ELSE IF** kullanarak koşul sayısı artırılabilir.

Dışarıdan alınan sayısal bir değere göre kontrol yapan bir **IF ... ELSE** tanımlayalım.

---

```
DECLARE @val INT;
SET    @val = 1;

IF @val = 1
    PRINT 'Bir'
ELSE IF(@val = 2)
    PRINT 'İki'
ELSE IF(@val = 3)
    PRINT 'Üç'
ELSE IF(@val >= 4) OR (@val <= 8)
BEGIN
    PRINT '4 - 8 arasında bir değer';
    PRINT '...'
END
ELSE
    PRINT 'Tanımlanamadı.';
```

---

Yukarıdaki sorguda, **IF ... ELSE** akış kontrolünün birçok özelliğini  **Messages** görebilirsiniz. **val** değişkenine aktarılan değeri **IF** blokları  **Bir** içerisinde kontrol ederek şartlara uyan ilk blok içerisindeki kodları çalıştırır.

- **val** değişkeninin değeri 1 olarak atandığında, ilk **IF** bloğu çalışacak ve **PRINT** ile ekrana 'Bir' yazacaktır.
- **val** değişkeninin değeri 4 ve 8 dahil, bu sayıların arasında bir değer ise, **BEGIN ... END** blokları arasındaki iki ayrı **PRINT** komutu çalışacaktır.
- Bu şartların hiç biri sağlanmadığı takdirde, en son olarak belirtilen **ELSE** bloğu çalışır.

En son oluşturulan **ELSE IF** bloğu içerisindeki **BEGIN ... END** bloğu dikkatinizi çekmiş olmalı. Bir blok içerisinde kodlama yaparken, blok içerisindeki kod satırı birden fazla olacak ise, bu durumu veritabanı motoruna bildirmek için **BEGIN ... END** blokları içerisine alınması gereklidir. **BEGIN ... END** bloklarının sorgu üzerindeki kullanımını kavrayabilmek için bu blok başlangıç ve bitiş komutlarını silerek sorguyu tekrar çalıştırın. **BEGIN** ve **END** komutlarını silerek çalıştırıldığınızda, sorgu çalışmayacak ve hata üretecektir.

Bir başka **IF ... ELSE** kuralı ise, hiç bir şartın sağlanmadığı durumda çalışması için kullanılan **ELSE** komutunun sorgunun en sonunda kullanılması zorunluluğudur.

**ELSE** komutunun sırasını değiştirerek herhangi bir sorgu sırasına yerleştirdiğinizde, sorguyu çalıştırmanıza gerek kalmadan, sorgunun altında, kırmızı bir hata çizgisi belirecektir. Sorguyu bu şekilde çalıştmak isterseniz de, bir hata üretilecektir.

**IF** koşulları içeresine, bir sorgu sonucunu değer olarak göndererek koşul sorgulaması yapılabilir.

## İÇ İÇE IF KULLANIMI

Bir çok işlemi genel olarak tek bir **IF ... ELSE** bloğu ile çözebiliyor olsak da, bazı durumlarda iç içe birden fazla IF kullanılması gerekebilir.

Parametre olarak verilen değeri iç içe **IF ... ELSE** kullanarak sorgulayacağız. Sayının büyüklüğüne göre, ekranda 'küçük', 'büyük' ya da 'ne küçük ne büyük' gibi bir ifade yazdıracağımız.

---

```
DECLARE @sayi INT;
SET    @sayi = 5;

IF @sayi > 100
    PRINT 'Bu sayı büyük。';
ELSE
BEGIN
    IF @sayi < 10
        PRINT 'Bu sayı küçük。';
    ELSE
        PRINT 'Ne küçük ne de büyük。';
END;
```

---



Yukarıdaki sorguda **sayı** değişkenine, çeşitli değer atamaları yapıldığında aşağıdaki sonuçları üretir.

- 10'dan küçük bir değer atandığında, '*Bu sayı küçük*' sonucunu üretir.
- 10 ile 100 sayılarına eşit ya da bu sayılar arasında ise, '*Ne küçük ne de büyük*' sonucunu üretir.
- 100'den büyük bir değer girilirse, '*Bu sayı büyük*' sonucunu üretir.

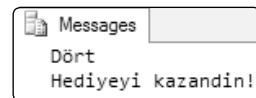
İç içe işlemler gerçekleştirmek için ihtiyacınız kadar **IF** kullanabilirsiniz.

Yukarıdaki sorguyu biraz daha geliştirerek 3 adet **IF** içeren bir sorgu oluşturalım.

---

```
DECLARE @sayi INT;
SET @sayi = 4;

IF @sayi > 100
    PRINT 'Bu sayı büyük.';
ELSE
BEGIN
    IF @sayi < 10
        IF @sayi = 1
            PRINT 'Bir'
        ELSE IF(@sayi = 2)
            PRINT 'İki'
        ELSE IF(@sayi = 3)
            PRINT 'Üç'
        ELSE IF(@sayi = 4)
BEGIN
            PRINT 'Dört'
            PRINT 'Hediyeyi kazandın!'
END
        ELSE IF(@sayi = 5)
            PRINT 'Beş'
        ELSE
            PRINT 'Bu sayı, 5 ve 10 arasında bir değere
sahip。';
    ELSE
        PRINT 'Ne küçük ne de büyük。';
END;
```



**sayı** değişkenine 4 değeri atadığınızda bir hediye kazanacaksınız!

Sorgularda dikkat edilmesi gereken konu, sorgunun basitliği ve işlevselliğidir. Bu tür iç içe sorgular sizinizi gördüğü ölçüde işlevsel, anlayabildiğiniz kadar basittir. İç içe sorgular kullanırken açıklama satırları kullanmaya özen göstermelisiniz. En iyi kod, kolay anlaşılabilir olmalıdır.

## CASE DEYİMİ

**CASE** ifadesi, programlama dillerinde de var olan akış kontrol özelliklerinden biridir. Akış kontrolü için kullanılan, neredeyse tüm teknikler genel olarak bir diğerinin yapabildiği işlemleri gerçekleştirebilir. Tabi ki aralarında farklılıklar vardır. Performans, kod okunabilirliği, karmaşık sorgular içerisinde kullanılabılır gibi seçim yapılması gerektiren durumlar olabilir.

**CASE** ifadesi, iki farklı şekilde kullanılabilir. Bunlar;

- Giriş ifadesi ile
- Boolean bir deyim ile

## SÖZ DİZİMİ (GİRİŞ DEYİMİ)

---

```
CASE <giriş deyimi>
WHEN <when deyimi> THEN <sonuç deyimi>
[...n]
[ELSE <sonuç deyimi>]
END
```

---

Giriş deyimi yöntemi ile kullanımda, **WHEN** koşulunda kullanılan değer ile karşılaştırma yapılacak olan bir giriş deyimi kullanılır.

## SÖZ DİZİMİ (BOOLEAN DEYİMİ)

---

```
CASE
WHEN <Boolean Deyimi> THEN <sonuç deyimi>
[...n]
[ELSE <sonuç deyimi>]
END
```

---

Boolean deyimi yöntemi ile kullanımda, her **WHEN** koşulunda **TRUE** ya da **FALSE** değeri verecek olan bir ifade sağlanır.

Satılan bisikletleri türleri (dağ, gezi vb.) ile birlikte listeleyelim.

---

```
SELECT ProductNumber, Category =
CASE ProductLine
    WHEN 'R' THEN 'Yol'
```

```

        WHEN 'M' THEN 'Dağ'
        WHEN 'T' THEN 'Gezi'
        WHEN 'S' THEN 'Diğer Satılıklar'
        ELSE 'Satılık Değil'

    END,
    Name
FROM Production.Product
ORDER BY ProductNumber;

```

---

	ProductNumber	Category	Name
1	AR-5381	Satılık Değil	Adjustable Race
2	BA-8327	Satılık Değil	Bearing Ball
3	BB-7421	Satılık Değil	LL Bottom Bracket
4	BB-8107	Satılık Değil	ML Bottom Bracket
5	BB-9108	Satılık Değil	HL Bottom Bracket
6	BC-M005	Dağ	Mountain Bottle Cage
7	BC-R205	Yol	Road Bottle Cage

Çalışanların görev, cinsiyet ve doğum tarihlerini listeleyelim.

---

```

SELECT JobTitle, BirthDate, Gender, Cinsiyet =
CASE
    WHEN Gender = 'M' THEN 'Erkek'
    WHEN Gender = 'F' THEN 'Kadın'
END
FROM HumanResources.Employee;

```

---

	Job Title	Birth Date	Gender	Cinsiyet
1	Chief Executive Officer	1963-03-02	M	Erkek
2	Vice President of Engineering	1965-09-01	F	Kadın
3	Engineering Manager	1968-12-13	M	Erkek
4	Senior Tool Designer	1969-01-23	M	Erkek
5	Design Engineer	1946-10-29	F	Kadın
6	Design Engineer	1953-04-11	M	Erkek
7	Research and Development Manager	1981-03-27	M	Erkek

## WHILE DÖNGÜSÜ

**WHILE** ifadesi, tüm programlama dillerinde olduğu gibi, döngüsel olarak bir koşul deyimini test etmek için kullanılır. Sonuç **TRUE** olduğu sürece, döngünün en başına dönerek tekrar test edilir. Sonuç **FALSE** olursa döngüden çıkarılır.

### Söz Dizimi:

---

```
WHILE Boolean_Deyim
    { sql_ifadesi | ifade_bloğu | BREAK | CONTINUE }
```

---

Değişken ve **WHILE** döngüsü içeren bir örnek hazırlayalım.

---

```
DECLARE @counter INT
SELECT @counter = 0

WHILE @counter < 5
BEGIN
    SELECT '@counter değeri : ' + CAST(@counter AS VARCHAR(1))
    SELECT @counter = @counter + 1
END;
```

---

	(No column name)
1	@counter degeri : 0
	(No column name)
1	@counter degeri : 1
	(No column name)
1	@counter degeri : 2
	(No column name)
1	@counter degeri : 3
	(No column name)
1	@counter degeri : 4

**WHILE** gibi döngü ifadeleri, bazen bir işlemi tekrarlayarak gerçekleştirmek için farklı amaçlarda kullanılabilir. Örneğin; bir tablo oluşturarak bu tabloya çok fazla sayıda test kaydı girilmesini istenebilir. Bu işlemi tek tek yapmak çok zaman alacaktır. Ancak bir döngü oluşturarak bir tabloya çok sayıda kayıt girilebilir.

---

```

CREATE TABLE #gecici(
    firmaID      INT NOT NULL IDENTITY(1,1),
    firma_isim   VARCHAR(20) NULL
);
WHILE (SELECT count(*) FROM #gecici) < 10
BEGIN
    INSERT #gecici VALUES ('dijibil'),('kodlab')
END;
SELECT * FROM #gecici;

```

---

	firmaID	firma_isim
1	1	dijibil
2	2	kodlab
3	3	dijibil
4	4	kodlab
5	5	dijibil
6	6	kodlab
7	7	dijibil
8	8	kodlab
9	9	dijibil
10	10	kodlab

Yukarıdaki soru ile birlikte `gecici` adında, geçici bir tablo oluşturulacak ve bu tabloya 10 adet kayıt girilecektir. Bu kayıtlar, `WHILE` döngüsü ile gerçekleştirilecektir.

10 yerine 100 ya da 1000 yazıldığında, `WHILE` döngüsü veri ekleme işlemini 100 ve 1000 kere gerçekleştirerek tabloya o sayıda kayıt ekleyecektir.

## BREAK KOMUTU

`WHILE` döngüsü sonlanmadan döngüden çıkmak için `BREAK` komutu kullanılır. `BREAK` komutu kullanmayı gerektirecek şekilde bir döngü yapısı oluşturulmamalı ve `BREAK` komutu kullanılmamalıdır. Ancak zorunlu hallerde bu komut kullanılmalıdır.

## CONTINUE KOMUTU

`BREAK` komutunun tam tersi bir işlem için kullanılır. Döngü içerisinde nerede olursanız olun, döngünün en başına dönmenizi sağlar.

Bir metni, ekranda istediğimiz kadar tekrar ettirerek yazdırıralım.

---

```
DECLARE @counter INT, @counter1 INT;
SELECT @counter = 0, @counter1 = 3;
WHILE @counter <> @counter1
BEGIN
    SELECT CAST(@counter AS VARCHAR) + ' : ' + 'dijibil.com & kodlab.com';
    SELECT @counter = @counter + 1;
END;
```

---

	(No column name)
1	0 : dijibil.com & kodlab.com
	(No column name)
1	1 : dijibil.com & kodlab.com
	(No column name)
1	2 : dijibil.com & kodlab.com

## **WAITFOR İFADESİ**

Belirli bir saatte belirlenen işlemi gerçekleştirmek için kullanılır.

### **Söz Dizimi:**

---

```
WAITFOR
DELAY 'zaman' | TIME 'zaman'
```

---

**WAITFOR** ifadesi, parametre olarak belirtilen sürenin dolmasını bekler. Süre dolduğu anda görevini gerçekleştirir.

## **WAITFOR DELAY**

**DELAY** parametresi, **WAITFOR** işlemi ile beklenen süreyi belirler. Saat, dakika ve saniye olarak değer verilebilir. En fazla 24 saat bekleme süresine sahiptir. Süre olarak gün değeri verilemez.

### **Söz Dizimi:**

---

```
WAITFOR DELAY '01:00'
```

---

1 dakika bekledikten sonra **sp\_helpdb** sistem prosedürüne çalışıralım.

```
BEGIN  
    WAITFOR DELAY '00:01';  
    EXECUTE sp_helpdb;  
END;
```

---

## WAITFOR TIME

**TIME** parametresi bir zaman belirtmek için kullanılır. Sadece saat cinsinden değer verilebilir. En fazla 24 saat değer verilebileceği gibi gün değeri verilemez.

İlk olarak **WAITFOR** ifadesinden önceki kodlar çalışır, **WAITFOR** ifadesine geldiğinde 11:00'a kadar beklenir. Saat eşleşmesi sağlandığında **WAITFOR** ifadesinden sonraki kodlar çalışır.

### Söz Dizimi:

---

```
WAITFOR TIME '11:00'
```

---

Belirli bir işlemi, belirlenen süre sonunda gerçekleştirecek bir örnek yapalım.

---

```
DECLARE @counter INT, @counter1 INT;  
SELECT @counter = 0, @counter1 = 3;  
WHILE @counter <> @counter1  
BEGIN  
    WAITFOR TIME '11:00'  
    SELECT CAST(@counter AS VARCHAR) + ' : ' + 'dijibil.com & kodlab.com';  
    SELECT @counter = @counter + 1;  
END;
```

---

## GOTO

Kod blokları arasında etiketleme yaparak akış sırasında farklı bir etikete işlem sırası atlatmak için kullanılır. Özel durumlar dışında kullanılan bir özellik değildir. Ancak bazı durumlarda kullanmak gerekebilir.

---

```
DECLARE @Counter int;  
SET @Counter = 1;  
WHILE @Counter < 10  
BEGIN
```

```
SELECT @Counter  
SET @Counter = @Counter + 1  
IF @Counter = 4 GOTO Etiket_Bir  
IF @Counter = 5 GOTO Etiket_Iki  
END  
Etiket_Bir:  
    SELECT 'Etiket 1'  
    GOTO Etiket_Uc;  
Etiket_Iki:  
    SELECT 'Etiket 2'  
Etiket_Uc:  
    SELECT 'Etiket 3'
```

---

	(No column name)
1	1
	(No column name)
1	2
	(No column name)
1	3
	(No column name)
1	Etiket 1
	(No column name)
1	Etiket 3



# SQL CURSOR'LARI

10

Cursor'ler, veri kümesini ele alarak her seferinde bir kayıt üzerinde işlem yapabilmeyi sağlayan yöntemdir. Kullanım olarak bir metin editörüne benzer. Metin editöründe imleç (*cursor*) hangi satırda ise, o satır üzerinde işlem yapabilirsiniz. Cursor'lar da aynı şekilde bir veri kümesi üzerinde ileri ve geri giderek satırlar üzerinde işlem yapmayı sağlar.

Cursor, varsayılan olarak sadece ileri doğru işlem yapar. Geriye doğru işlem yapabiliyor olsa da, ana kullanım yöntemi ileriye doğrudur. Ayrıca sadece ileri doğru çalışan Cursor ile, kaydırma yapabilen (ileride göreceğiz) Cursor'lar arasında önemli performans ve hız farkı vardır.

En hızlı Cursor, sadece ileri doğru okuma işlemi yapan Cursor'dür. Sadece ileri doğru okuma yapan Cursor'leri, DOT.NET geliştiricileri için `SqlDataReader` nesnesine benzetebiliriz. Her ikisi de sadece ileri doğru okuma yaptıkları ve önceki satırlarla ilgilenmediği için çok hızlıdır.

Veritabanında bir `SELECT` sorgusu ile alınan kayıtlar üzerinde, döngü yapısı oluşturarak tüm satırları tek tek inceleyebilmek, üzerinde işlem yapabilmek için Cursor programlanmalıdır. Cursor'ler veritabanında saklanmazlar. Cursor'ler kitap içerisinde işlenen Stored Procedure'ler içerisinde kullanılmalıdır.

## ISO Söz Dizimi:

---

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[ ; ]
```

---

### **Transact-SQL Genişletilmiş Söz Dizimi:**

---

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

---

## **CURSOR İÇERİSİNDEKİ SELECT SORGUSUNUN FARKLARI**

Cursor içerisinde oluşturulan **SELECT** sorgusu her ne kadar aynı söz dizimine sahip olsa da Cursor'lere özel bazı farklılıklar vardır.

- Cursor ve sonuç kümesi bildirim sırasında isimlendirilir, daha sonra bu isimler kullanılır.
- Cursor bildirimi uygulamadan ayrı olarak yapılır.
- Cursor, kendi içerisinde açık ve kapalı olma durumuna sahiptir. Siz kapatana kadar Cursor açık kalır.
- Siz silmeden Cursor hafızadan silinmez.
- Cursor, işlevsel yeteneklere sahip olabilmesi için bazı özel komutları içerir.

## **CURSOR'LER NEDEN KULLANILIR?**

Cursor'ler veritabanında sürekli kullanmanız gereken bir özellik olmasa da, gerekiği durumlarda birçok faydalı işlevle sahiptir. Az da olsa kullanmanız gereken faydalı bir özelliktir.

Cursor'ler genel olarak şu amaç ile kullanılır;

- Bir sorgu sonucunda (resultset) bulunan kayıtlar içerisinde gezinmek, önceki ya da sonraki kayda gitmek, ilk ya da son kayda gitmek istenildiğinde kullanılır.
- Sorgu sonucu dönen değerleri incelemeye tabi tutarak, üzerinde değişiklik yapılması gereken satırlarda güncelleme işlemi yapmak.

- Diğer kullanıcılar tarafından yapılan değişikliklerin görünürlük seviyesini ayarlamak için kullanılır. (Görünürlük seviyesi konusunu daha sonra inceleyeceğiz)
- Trigger ya da Stored Procedure'lerin bir sorgu sonucuna satır satır erişmesini sağlamak.

## CURSOR'UN ÖMRÜ

Cursor birden fazla parçadan oluşur. Cursor'un sistem üzerindeki yaşam döngüsünü öğrenmeden ömrünü kavramak güçtür. Bu nedenle, Cursor ömrünü anlayabilmek için bir Cursor oluşturalım.

Bir Cursor geliştirme kısımları şu şekildedir;

- Bildirim
- Açılış
- Kullanım/Yönlendirme
- Kapanış
- Hafızada Ayrılan Belleği Boşaltmak

Cursor'lerin çalışma modelini kavramak için karmaşık değil, basit bir sorgu ile ilk örneği gerçekleştirmeliyiz.

Ürünler tablosunda **ProductID** ve **Name** sütunlarını Cursor ile listeleyelim.

---

```

DECLARE @ProductID INT
DECLARE @Name VARCHAR(255)
DECLARE ProductCursor CURSOR FOR
    SELECT ProductID, Name FROM Production.Product
OPEN ProductCursor
FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END
CLOSE ProductCursor
DEALLOCATE ProductCursor

```

---

Sorgu çalıştırıldığında aşağıdaki gibi bir listeleme gerçekleştirmeli.

```

1 - Adjustable Race
2 - Bearing Ball
3 - BB Ball Bearing
4 - Headset Ball Bearings
316 - Blade
317 - LL Crankarm
318 - ML Crankarm
319 - HL Crankarm
320 - Chainring Bolts
321 - Chainring Nut
322 - Chainring

```

Cursor başarılı bir şekilde çalıştı. Ancak yukarıdaki komutları henüz incelemedik. Şimdi, oluşturduğumuz Cursor'ü yukarıda belirttiğimiz geliştirme kısımlarına göre sınıflandırarak inceleyelim.

## BİLDİRİM

Bir Cursor'ün geliştirme kısımlarında ilk sırada bildirim kısmı olduğunu belirtmiştim. Cursor içerisinde kullanılacak değişkenlerin tanımlanıldığı, yani bildirimlerinin gerçekleştiği bölümdür. Bu bildirimler, Cursor ile `FETCH` edilecek kayıtların hafızada tutularak Cursor'ün üzerinde bulunduğu kayıtları temsil eder. Cursor satır satır okuduğu veriyi bu değişkenlere aktararak üzerinde işlem yapılabilmesini sağlar.

Cursor'ün kendisi de değişkenler ile birlikte burada tanımlanır. Değişkenler ile Cursor'ün tanımlanması arasındaki tek farkı Cursor'ün `@` işaretini ile başlayan bir isimlendirmeye sahip olmamasıdır. Cursor isminden sonra `CURSOR FOR` kullanılarak Cursor'ün kullanacağı veriyi temsil eden sorgu oluşturulurur.

---

```

DECLARE @ProductID INT
DECLARE @Name  VARCHAR(255)
DECLARE ProductCursor CURSOR FOR
    SELECT ProductID, Name FROM Production.Product;

```

---

## AÇILIS

Cursor oluşturulduktan sonra veri okuma işlemine hazır hale gelmesi için **OPEN** komutu ile açılması gereklidir. Açılmış kısmı, Cursor'ün **OPEN** ile açıldığı kısmı temsil eder. Açılan Cursor, kaynak tüketimine başlamış demektir. Kapanış kısmında anlattığımız gibi Cursor işlemi sonunda açık Cursor kapatılmalıdır.

---

**OPEN ProductCursor**

---

## KULLANIM / YÖNLENDİRME

Cursor işleminin yönetim kısmıdır. Bir Cursor ile yapılması gereken işlemler, verinin yönetilmesi işlemleri bu kısımda gerçekleştirilir. Bildirim kısmında bildirimi yapılan değişkenlere bu bölümde **WHILE** döngüsü ile elde edilen kayıt verileri aktarılır. Ekrana yazdırma ya da benzeri işlemler de bu kısımda gerçekleştirilir.

Aşağıdaki sorguda, **ProductCursor** isimli Cursor sorunsuz çalışıyor ise **WHILE** döngüsü içerisine girilir ve elde edilen veriler **PRINT** ile ekranda listelenir.

İlk kaydın yakalanması için bir **FETCH** işlemi gerçekleştirilir. İlk **FETCH** ile bir değer bulunur ise, içerisindeki veriler **FETCH**'deki değişkenlere aktarılır. Bu işlemden sonra **WHILE** döngüsü gerçekleştirir.

---

```

FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END

```

---

**WHILE** döngüsüne girmeden önce Cursor'de veri olduğunu nasıl anlıyoruz?

**@@FETCH\_STATUS** global değişken değeri, her satır getirildiğinde güncellenir. Eğer değer 0 (*sıfır*) ise **FETCH** işlemi başarılıdır ve **WHILE** döngüsüne girilir. Daha sonra **FETCH** işlemi döngü halinde gerçekleştirir.

**@@FETCH\_STATUS** isimli global değişken **FETCH** işleminin sonucunu bize 3 değer ile bildirir.

- 0 : **FETCH** işlemi başarılıdır. Kayıt vardır.

- 1 : Kayıt bulunamadı.
- 2 : **FETCH** başarısız. Bu hata, son kaydın ötesinde ya da ilk kaydın da öncesinde bulunulduğunu belirtir.

## KAPANIŞ

Cursor açılışı gerçekleştikten sonra **CLOSE** ile kapatılmadığı sürece sürekli açık kalacaktır. Açık kaldığı takdirde Cursor'ün kullanıldığı yerlerde birçok hata ile karşılaşılabilceğ gibi, sürekli açık kalan Cursor sistem kaynağını gereksiz şekilde tüketecektir. Cursor yönetimi açısından, açık olması gereken özel durumlar haricinde mutlaka kapatılması gereklidir. Kapanış işlemiyle birlikte Cursor ile ilgili kilitler serbest bırakılır.

---

```
CLOSE ProductCursor
```

---

## HAFIZADA AYRILAN BELLEĞİ BOŞALTMAK

**CLOSE** komutu, Cursor'ün kapanmasını sağlar. Ancak Cursor'ün kapanması demek kullandığı hafızanın boşaltılması anlamına gelmez. Hafiza boşaltma işleminin gerçekleşmesi için **CLOSE** işleminden sonra **DEALLOCATE** komutu kullanılmalıdır.

---

```
DEALLOCATE ProductCursor
```

---

## CURSOR TIPLERİ VE ÖZELLİKLERİ

Cursor'ler işlev ve yeteneklerine göre farklı tip ve özelliklerle ayrırlar. Cursor'ların özellik ve tiplerini inceleyelim.

### SCOPE

Scope, Cursor'lerin görünebilirlik ayarını belirtir. **LOCAL** ve **GLOBAL** olarak ikiye ayrılır. En kısa tanım ile şu şekilde özetlenebilir. Bir sproc içerisinde oluşturulan Cursor **GLOBAL** scope'u içerisinde ise, bir başka sproc içerisinde bu Cursor'e erişilebilir. Ancak bu Cursor **LOCAL** scope içerisinde ise bir başka sproc içerisinde erişilemez. **GLOBAL** olarak tanımlanan bir Cursor ismi ile başka bir sproc içerisinde dahi olsa yeni bir Cursor tanımlanamaz. Etki alanı çerçevesinde benzersiz isime sahip olmalıdır. **GLOBAL** scope içerisindeki Cursor'de başka bir sproc içerisinde de erişilebilir olduğu için, farklı sproc içerisinde de olsa, aynı isimde tanımlama işlemi hataya yol açar.

SQL Server Cursor'leri varsayılan olarak **GLOBAL** özelliğindedir. SQL Server'da bir şeyin **LOCAL** ya da **GLOBAL** olması, tüm bağlantılar ya da sadece geçerli bağlantılar tarafından görülmemesini ifade eder. Yukarıda, bir sproc içerisindeki Cursor'e bir diğerinden ulaşma durumu, bu **görülme** olayını ifade eder.

## KAYDIRILABILIRLIK

Kaydırılabilirlik (*scrollability*) özelliği, SQL Server'da Cursor'lerin hareket yönünü ifade eder. Varsayılan olarak sadece ileriye doğru hareket eden Cursor'ler kaydırılabilirlik seçenekleri ile ileri ve geri hareket kabiliyetlerinin kullanılmasını sağlar.

## **FORWARD\_ONLY** ÖZELLİĞİ

Cursor'ün varsayılan yöntemidir. Sadece ileriye doğru hareket eder. Tek yöne sahip ve sadece ileriye doğru çalıştığı için kullanılabilecek gezinme özelliği **FETCH NEXT**'dir.

Daha önce örnek verdığımız DOT.NET içerisindeki **DataReader** nesnesi gibi çalışır. Hatırlarsanız, **DataReader**'da sadece ileriye doğru çalışır ve bu nedenle çok hızlı sonuç üretirdi. Ancak **DataReader**'da gösterilen bir kaydı tekrar elde etmek için geriye doğru gidemezdik. Aynı kaydı tekrar elde etmek için sorgunun tekrar çalıştırılması gerekiyordu. **FETCH NEXT** de aynı şekilde çalışır. Cursor bir sonraki kayda geçtikten sonra önceki kayda tekrar ulaşmak için Cursor'ün kapatılıp yeniden açılması gerekir.

## **SCROLLABLE** ÖZELLİĞİ

Cursor'ü ileri geri hareket ettirmek için kullanılır. Kaydırma işleminin temel özelliği **FETCH** anahtar sözcüğüdür.

- **FETCH** ile kullanılan yan komutlar şu şekildedir:
- **NEXT** : Bir sonraki kayda geçer.
- **PRIOR** : Bir önceki kayda geçer.
- **FIRST** : İlk kayda geçer.
- **LAST** : Son kayda geçer.

Normal bir Cursor tanımlamasına ek olarak **SCROLL** ifadesi içerir.

**Örnek:**


---

```
CREATE PROC proc_name
AS
DECLARE @var1 INT, @var1 INT
DECLARE cursor_name CURSOR
LOCAL
SCROLL
FOR
SELECT ...
```

---

**DUYARLILIK KAVRAMI**

Cursor duyarlılığı, Cursor açıldıktan sonra veritabanında gerçekleşen değişiklıkların dikkate alınıp alınmadığı konusu ile ilgilidir. Bazı durumlarda, kullanılacak Cursor'ün veritabanı değişikliklerini dikkate almamasını ve veriler değişse de çalışmasına statik veriler ile devam etmesi istenebilir. Bu durumda statik Cursor kullanılır. Statik Cursor, oluşturulduktan sonra veritabanındaki değişiklıklere karşı duyarsız olacaktır. Ancak statik Cursor'ün aksine, dinamik Cursor veritabanında gerçekleşen ekleme, güncelleme, silme gibi değişiklik işlemlerine karşı duyarlı olacaktır.

**CURSOR'LERLE SATIRLARI****DOLAŞMAK: FETCH**

**FETCH** komutu, Cursor için gerçek anlamda gezinme işini yapan komuttur. Bir Cursor'ün üzerinde, ileri ya da geri herhangi bir hareket ve konumlandırma işlemi yapılacak ise, bu işi yapacak olan komuttur.

**FETCH** işlemi ile kullanılan komutlarla ilgili birçok örnek yaptık. Öncelik sırasında sonlarda olan bazı komutlar için ise bu bölümde örnekler yapacağız.

Çeşitli örnekler yapmak için bir Cursor tanımlayalım.

---

```
DECLARE ProductCursor SCROLL CURSOR FOR
SELECT ProductID, Name FROM Production.Product WHERE ProductID < 5
```

---

Cursor'ü açalım.

---

```
OPEN productCursor
```

---

Cursor artık çalışıyor. Artık sırasıyla, tüm işlemleri deneyebiliriz.

## FETCH NEXT

Cursor ile ilgili hazırladığımız bir çok örnekte **NEXT** komutunu kullandık. **FETCH** işleminde en çok ve ileri doğru hareket için kullanılan sorgu ifadesidir.

**NEXT**, sonuç kümesinde bir satır ileri girmeyi sağlar. Cursor bildirimini **FORWARD\_ONLY** yapıldı ise **NEXT** ifadesi kullanılır.

---

-- Bir kayır ileri hareket ettirir.

```
FETCH NEXT FROM ProductCursor
```

---

## FETCH PRIOR

**NEXT** ifadesinin tersi olarak çalışır. Cursor'ün bulunduğu kayıttan geriye doğru bir kayıt hareket etmeyi sağlar. Yani, "önceki" anlamına gelir.

---

- Önceki kayda gider.

```
FETCH PRIOR FROM ProductCursor
```

---

## FETCH FIRST

**FIRST** ifadesi ilk kayda geri dönmek için kullanılır. Cursor hangi satırda olursa olsun, **FIRST** ilk satıra dönmesini sağlar.

---

-- İlk kayda gider

```
FETCH FIRST FROM ProductCursor
```

---

## FETCH LAST

**LAST** ifadesinin tersi olarak çalışır. Cursor'ün son kayıt üzerine gitmesini sağlar. Cursor hangi satırda olursa olsun, son kayıt üzerinde konumlanmak için **LAST** kullanılır.

---

-- Son kayda gider

```
FETCH LAST FROM ProductCursor
```

---

## FETCH RELATIVE

Cursor'un bulunduğu satırdan belirli sayıda satır kadar ileri ya da geriye doğru hareket etmek için kullanılır.

-- Bulunan yerden 3 kayıt ileri konumlanır.

```
FETCH RELATIVE 3 FROM ProductCursor
```

-- Bulunan kayıttan 2 kayıt geriye konumlanır.

```
FETCH RELATIVE -2 FROM ProductCursor
```

## FETCH ABSOLUTE

Cursor başlangıcından itibaren ne kadar satır istendiğini belirten bir `Integer` değer alır. Belirtilen değer pozitif ise Cursor'un başından, negatif ise sonundan itibaren verilen değer kadar satır anlamına gelir.

-- Baştan 2. kayda konumlanır

```
FETCH ABSOLUTE 2 FROM ProductCursor
```

Tüm işlemlerden sonra Cursor'ü kapatalım ve hafızadaki alanı boşaltalım.

```
CLOSE ProductCursor
```

```
DEALLOCATE ProductCursor
```

## TYPE\_WARNING

Cursor üzerinde yapılacak kapalı dönüştürme işlemi gerçekleştirken uyarı mesajı verir. Örneğin; Keyset-Driven Cursor tanımlandığında, eğer üzerinde unique indeks yoksa, SQL Server otomatik olarak Keyset-Driven'i Statik Cursor'e dönüştürür. Bu tür dönüşümler kapalı dönüşüm olarak bilinir.

Bir örnek üzerinde çalışalım.



Aşağıdaki sorguyu çalıştırmadan, daha önce açık bulunan ProductCursor isimli Cursor halen açık ise kapatın ve hafızadan boşaltın.

```
DECLARE ProductCursor CURSOR
```

```
GLOBAL
```

```
SCROLL
```

```
KEYSET
```

```

TYPE_WARNING -- Dönüşürme mesajını verecek olan özellik
FOR
SELECT ProductID, Name FROM Production.Product

DECLARE @ProductID INT
DECLARE @Name      VARCHAR(5)

OPEN ProductCursor

FETCH NEXT FROM ProductCursor INTO @ProductID, @Name

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END

CLOSE ProductCursor
DEALLOCATE ProductCursor

```

1	Adjus
2	Beari
3	BB Ba
4	Heads
316	Blade
317	LL Cr
318	ML Cr
319	HL Cr
320	Chain
321	Chain
322	Chain

Diğer Cursor örneklerimizden farklı olan tek kısım **TYPE\_WARNING** tanımlaması oldu. Soru sonucunda bir mesaj alındı. Bu bir hata mesajı değil, sadece bilgilendirme amaçlı uyarı mesajıdır.

## CURSOR TIPLERİ

Cursor'ler çeşitli özelliklerine göre 4 farklı tipe ayrılır.

- Statik
- Keyset ile Çalıştırılan (*Keyset-Driven*)
- Dinamik
- Sadece İleri (*Forward-Only*)

Dört tip olarak ayrılan Cursor'lerin aralarındaki farklılığın sebepleri kaydırılabilirlik ve veritabanındaki değişikliklere karşı duyarlılıklarıdır.

### STATİK CURSOR'LAR

Cursor oluşturulduktan sonra hiç bir şekilde değiştirilmeyen, yani statik olarak çalışan Cursor tipidir. Statik Cursor oluşturulduğunda, **tempdb** veritabanında, geçici tablonun tutabileceği alan miktarında kayıt tutulur.

Statik Cursor, `tempdb` üzerinde tutulur ve bir geçici tablo görevi görür. Statik Cursor içerisindeki kayıtlarda güncelleme yapılamaz. Bazı nesne modelleri güncellemeye olanak tanısa da sonuç olarak değiştirilen veri veritabanına yazılamaz.

Statik Cursor özelliğini örnek üzerinde inceleyelim.

Cursor işleminde kullanacağımız ve üzerinde değişiklik yapacağımız tabloyu oluşturalım.

---

```
SELECT ProductID, Name INTO ProductCursorTable FROM Production.Product;
```

---

Cursor bildirimini yapalım.

---

```
DECLARE ProductCursor CURSOR
GLOBAL -- Cursor batch dışında da kullanılabilir.
SCROLL -- Geriye doğru kaydırma yapılabilir.
STATIC
FOR
SELECT ProductID, Name FROM ProductCursorTable
```

---

Cursor için kullanılacak değişkenlerin bildirimini yapalım.

---

```
DECLARE @ProductID INT
DECLARE @Name      VARCHAR(30)
```

---

Cursor'ü açalım.

---

```
OPEN ProductCursor
```

---

Cursor ile ilk kaydı seçelim.

---

```
FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
```

---

Cursor'deki tüm kayıtları seçmek için bir döngü oluşturalım.

---

```
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END;
```

---

Cursor, sorgu sonucunda bize ilgili tablodaki kayıtları listeledi.

```
1 - Adjustable Race
879 - All-Purpose Bike Stand
712 - AWC Logo Cap
3 - BB Ball Bearing
2 - Bearing Ball
877 - Bike Wash - Dissolver
316 - Blade
843 - Cable Lock
952 - Chain
324 - Chain Stays
322 - Chainring
320 - Chainring Bolts
```

Şimdi, oluşturduğumuz tablo ve Cursor'ü kullanarak bir güncelleme işlemi yapalım ve statik Cursor'lerin nasıl çalıştığını örnek ile inceleyelim.

**ProductCursorTable** tablomuzdaki içeriği listeleyelim.

---

```
SELECT ProductID, Name FROM ProductCursorTable;
```

---

	ProductID	Name
1	1	Adjustable Race
2	879	All-Purpose Bike Stand
3	712	AWC Logo Cap
4	3	BB Ball Bearing
5	2	Bearing Ball
6	877	Bike Wash - Dissolver
7	316	Blade

Cursor'ün kaynak olarak kullandığı tabloda bir güncelleme yapalım.

---

```
UPDATE ProductCursorTable
SET Name = 'Update ile değiştirildi'
WHERE ProductID = 1;
```

---

Tablo üzerinde güncelleme işlemi başarılı.

Şimdi, oluşturduğumuz Cursor'ü tekrar çalıştırarak verinin Cursor'de değişip değişmediğini görelim.

---

```

DECLARE @ProductID INT
DECLARE @Name      VARCHAR(30)

FETCH FIRST FROM ProductCursor INTO @ProductID, @Name

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END

```

---

```

1 - Adjustable Race
879 - All-Purpose Bike Stand
712 - AWC Logo Cap
3 - BB Ball Bearing
2 - Bearing Ball
877 - Bike Wash - Dissolver
316 - Blade
843 - Cable Lock
952 - Chain
324 - Chain Stays
322 - Chainring

```

Sorgu sonucunda, tabloda yapılan değişikliğin Cursor tarafından fark edilmediğini görüyoruz. Durumu onaylamak için tabloyu tekrar listeleyelim. Tabloda değişmiş olan veri, Cursor'de değişmemiştir.

---

```
SELECT ProductID, Name FROM ProductCursorTable;
```

---

	ProductID	Name
1	1	Update ile degistirildi
2	879	All-Purpose Bike Stand
3	712	AWC Logo Cap
4	3	BB Ball Bearing
5	2	Bearing Ball
6	877	Bike Wash - Dissolver
7	316	Blade

Son işlem olarak Cursor kapatılmalı ve hafızada kapladığı yer boşaltılmalıdır.

---

```
CLOSE ProductCursor
DEALLOCATE ProductCursor
```

---

## Örnekte kullanılan kodların tamamı;

---

```
SELECT ProductID, Name INTO ProductCursorTable FROM Production.Product
DECLARE ProductCursor CURSOR
GLOBAL
SCROLL
STATIC
FOR
SELECT ProductID, Name FROM ProductCursorTable
DECLARE @ProductID INT
DECLARE @Name      VARCHAR(30)
OPEN ProductCursor

FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END
UPDATE ProductCursorTable
SET Name = 'Update ile değiştirildi'
WHERE ProductID = 1
FETCH FIRST FROM ProductCursor INTO @ProductID, @Name
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END
CLOSE ProductCursor
DEALLOCATE ProductCursor
```

---

Yukarıdaki sorguda neler yapıldı?

- Bir tablo oluşturularak içerisine veri eklendi.
- Oluşturulan Static Cursor ile bu tablodaki veriler elde edildi. Ekranda listeleme yapıldı.
- Tablo üzerinde güncelleme işlemi yapılarak gerçek tablodaki veri gerçek anlamda değiştirildi.
- Statik Cursor'ler oluşturulduktan sonra veritabanı ile bağlantısını kestiği için, yapılan güncellemenin farkında olamadı. Bu nedenle Cursor'deki veriler değişmedi.

## ANAHTAR TAKIMI İLE ÇALIŞTIRILAN CURSOR'LER

Anahtar Takımı Cursor'ler (*Keyset-Driven Cursor*), veritabanındaki tüm satırları **unique** olarak tanımlayan veri kümesi sağlayan Cursor'lerdir.

Anahtar Takımı ile Çalıştırılan Cursor'lerin Özellikleri:

- Cursor'ün kullandığı tablo üzerinde unique indeks olması gereklidir.
- tempdb veritabanında veri kümeleri değil, sadece anahtar takımı saklanır.
- Satırlarda meydana gelen tüm değişikliklere duyarlıdır. Ancak Cursor oluşturulduktan sonra eklenen yeni satırlara karşı duyarlı değildir.

Anahtar takımı ile çalışan Cursor'ler kullanıldığında, tüm anahtarlar tempdb içerisinde saklanır. Bu anahtarlar gerçek veriye ulaşmak için, veri bulma yöntemi olarak kullanılır. Bir işaretleyici olarak da düşünülebilir. **FETCH** işlemi sırasında ulaşılacak gerçek veriye bu anahtarlar ile ulaşılır.

Şimdi bir Keyset-Driven Cursor oluşturalım.



Bu örnekte **ProductCursorTable** tablosunu ve **ProductCursor** isimli Cursor'ü tekrar kullanacağız. Bu nedenle daha önce kullandığımız **ProductCursorTable** tablosunu **DROP TABLE** ifadesi ile silin ve cursor'ü de **DEALLOCATE** ile hafızadan boşaltın.

Cursor kullanacağımız tabloyu oluşturalım.

---

```
SELECT ProductID, Name
INTO ProductCursorTable
FROM Production.Product WHERE ProductID > 0 AND ProductID < 300
```

---

Cursor tablosu üzerinde bir **Primary Key** biçiminde **Unique** indeks oluşturalım.

---

```
ALTER TABLE ProductCursorTable
ADD CONSTRAINT PKCursor
PRIMARY KEY(ProductID)
```

---

**Identity** sütunlarda boş olan sütunları değiştirebilmek için;

---

```
SET IDENTITY_INSERT ProductCursorTable ON;
```

---

## Cursor bildirimini yapalım.

---

```
DECLARE ProductCursor CURSOR
GLOBAL
SCROLL
KEYSET
FOR
SELECT ProductID, Name FROM Production.Product
WHERE ProductID > 0 AND ProductID < 300
```

---

## Cursor'de kullanılacak değişkenleri tanımlayalım.

---

```
DECLARE @ProductID INT
DECLARE @Name      VARCHAR(30)
```

---

## Cursor'ü açalım.

---

```
OPEN ProductCursor
```

---

## İlk kaydı bulup getirelim.

---

```
FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
```

---

## Tüm kayıtları bulup getirmek için bir döngü oluşturalım.

---

```
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END
```

---

```
1 Adjustable Race
2 Bearing Ball
3 BB Ball Bearing
4 Headset Ball Bearings
```

## Cursor'ün kullandığı tabloda güncelleştirme yapalım.

---

```
UPDATE ProductCursorTable
SET Name = 'Değiştirildi'
WHERE ProductID = 1
```

---

Cursor'ün kullandığı tabloda silme işlemi yapalım.

---

```
DELETE ProductCursorTable
WHERE ProductID = 2
```

---

Cursor'ün kullandığı tabloda kayıt ekleme işlemi yapalım.

---

```
INSERT INTO ProductCursorTable(ProductID, Name)
VALUES(299,'Test Veri')
```

---

İlk kaydı bulalım.

---

```
DECLARE @ProductID INT;
DECLARE @Name    VARCHAR(30);
FETCH FIRST FROM ProductCursor INTO @ProductID, @Name
```

---

Bir **WHILE** döngüsü oluşturarak verilerin, değiştirme (**Insert**, **Update**, **Delete**) işlemlerinden sonraki durumlarını inceleyelim.

---

```
DECLARE @ProductID INT;
DECLARE @Name    VARCHAR(30);
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@ProductID AS VARCHAR) + ' ' + @Name
    FETCH NEXT FROM ProductCursor INTO @ProductID, @Name
END
```

---

```
2 Bearing Ball
3 BB Ball Bearing
4 Headset Ball Bearings
```

Cursor'ü kapatalım ve hafızada kapladığı yeri boşaltalım.

---

```
CLOSE ProductCursor
DEALLOCATE ProductCursor
```

---

Cursor'lerin aralarındaki teknik farklılıklarını kavramanın en iyi yolu farklı işlemler için farklı Cursor'ler programlamaktır.

## DİNAMİK CURSOR'LER

Temeldeki veride yapılan değişiklikleri proaktif olarak göstermezler. Ancak, tüm değişikliklere karşı duyarlı olmaları nedeniyle dinamik olarak adlandırılırlar.

Cursor oluşturulmasından sonra eklenen kayıtlar, güncellenen satırların Cursor'da da güncellenmesi, silinen kayıtların Cursor'den silinmesi işlemlerine karşı duyarlıdır. Dinamik Cursor'lerin performansı düşürdüğü durumlar olabilir. Eğer temel tablodaki kayıtlar çok geniş değilse dinamik Cursor'ler performanslı çalışmaya devam edecektir. Ancak, temel tablonun veri ve sütun genişliği fazla ise performanslı çalışmayacek ve bu durum bir sorun oluşturacaktır. Bunun nedenini anlamak için dinamik Cursor'lerin çalışmasını incelemek gereklidir.

Diğer Cursor tiplerinde Cursor oluşturulduktan sonra veri hafızaya alınır ve üzerinde okuma işlemi gerçekleştiriliyordu. Bu nedenle, bir veri kümesi Cursor'e alındıktan sonra tablodaki yeni kayıt ve güncellenen kayıtlardan haberi olmazdı. Ancak dinamik Cursor'ler tablo temelinde veri duyarlılığını sahiptir. Bunun sebebi, dinamik Cursor'lerin her `FETCH` komutu çalıştırıldığında Cursor'ü yeniden oluşturmaları ve içerdeği `SELECT` ifadesini `WHERE` ile birlikte tekrar çalıştırmasıdır.

Bazen sorgu işlemlerinde hız her şey demek değildir. Dinamik cursor'ler genellikle cache üzerinde çalışır. Yani, RAM (*bellek*) kullanımı yüksek bir cursor tipidir. Hatırlarsanız Keyset Cursor'ler ise `tempdb` üzerinden, yani disk üzerinden çalışıyordu. Geniş bellek kapasitesinin olduğu durumlarda RAM üzerinden çalışmak, disk'e göre daha hızlıdır. Ancak, SQL Server'a yoğun sorgu oluşturulduğunda, geniş tablo yapısı ve veriye sahip olduğu durumlarda, SQL Server'in kullandığı bellek alanı artacağı için sistemdeki kullanılmayan bellek alanı düşecektir. Bu da RAM üzerinde çalışan dinamik Cursor'ler için büyük risktir. Gene de, küçük veri kümeleri ile çalışılıyorsa, hem veri duyarlılığı olması hem de performanslı çalışmaları nedeniyle dinamik Cursor'ler tercih edilebilirler.

Aşağıdaki şekilde tanımlanırlar.

---

```
DECLARE cursor_ad CURSOR
DYNAMIC -- Cursor'e dinamik özelliği kazandırır.
SCROLL -- Cursor'ile ileri ve geri hareket özelliği kazandırır.
```

GLOBAL -- Cursor'e batch dışında kullanılabilme özelliği kazandırır.

FOR

select\_ifadesi

---

## FOR <SELECT>

Cursor bildirimlerinde gördüğünüz gibi, en önemli Cursor komutu parçasıdır. Bir Cursor için gerekli **SELECT** cümleciğinin tanımlanacağı kısımdır.

## FOR UPDATE

**FOR UPDATE** özelliği, Cursor içerisinde belirtilen sütunların güncellenmesi için kullanılır. Sadece sütun listesinde bulunan sütunlar güncellenebilir. Sütun listesinde bulunmayanlar salt okunur kalacaktır.

**FOR UPDATE** için bir örnek yapalım.

Güncellemeye için kullanacağımız tabloyu oluşturalım.

---

```
CREATE TABLE dbo.Employees (
    EmployeeID INT NOT NULL,
    Random_No  VARCHAR(50) NULL
) ON [PRIMARY]
```

---

Employees tablosunun içeriğini listeleyelim. Boş, yani herhangi bir kayıt eklemedik.

EmployeeID	Random_No

**WHILE** döngüsü için gerekli tanımlamaları yapalım ve döngüyü çalışıralım.

---

```
SET NOCOUNT ON
DECLARE @Rec_ID AS INT
SET @Rec_ID = 1

WHILE (@Rec_ID <= 100)
BEGIN
    INSERT INTO Employees
    SELECT @Rec_ID, NULL
    IF(@Rec_ID <= 100)
        BEGIN
```

```

SET @Rec_ID = @Rec_ID + 1
CONTINUE
END
ELSE
BEGIN
BREAK
END
END
SET NOCOUNT OFF

```

---

Yukarıdaki **WHILE** döngüsü ile birlikte, artık 100 kayıtlık bir tablomuz oluştu. 100 kayıt oluşturmak için **WHILE** döngüsü kullanıldı.

Tablodaki veriyi listelemek için;

---

```
SELECT EmployeeID, Random_No FROM Employees;
```

---

	EmployeeID	Random_No
1	1	NULL
2	2	NULL
3	3	NULL
4	4	NULL
5	5	NULL
6	6	NULL
7	7	NULL

Tabloya bir Constraint ekliyoruz.

---

```

ALTER TABLE dbo.Employees
ADD CONSTRAINT PK_Employees PRIMARY KEY CLUSTERED
(
    EmployeeID ASC
) ON [PRIMARY]

```

---

Güncelleme işlemini gerçekleştirelim. Bu kısımda, daha önce oluşturduğumuz tablonun **NULL** bırakılan **Random\_No** sütununa rastgele oluşturulan değerler atayarak güncelleme gerçekleştireceğiz.

```
SET NOCOUNT ON

DECLARE
    @Employee_ID      INT,
    @Random_No        VARCHAR(50),
    @TEMP              VARCHAR(50)

DECLARE EmployeeCursor CURSOR FOR
SELECT EmployeeID, Random_No FROM Employees FOR UPDATE OF Random_No
OPEN EmployeeCursor
FETCH NEXT FROM EmployeeCursor
INTO @Employee_ID, @Random_No
WHILE (@@FETCH_STATUS = 0)
BEGIN
    SELECT @TEMP = FLOOR(RAND()*1000000000000000)
    UPDATE Employees SET Random_No = @TEMP WHERE CURRENT OF
EmployeeCursor
    FETCH NEXT FROM EmployeeCursor
    INTO @Employee_ID, @Random_No
END
CLOSE EmployeeCursor
DEALLOCATE EmployeeCursor

SET NOCOUNT OFF
```

---

**FOR UPDATE** ile eklenen **Random** verilerden sonra tablo kayıtlarını tekrar listeleyelim.

---

```
SELECT EmployeeID, Random_No FROM Employees;
```

---

	EmployeeID	Random_No
1	1	1.61579e+012
2	2	2.2216e+012
3	3	1.8678e+012
4	4	7.76989e+012
5	5	5.66375e+012
6	6	5.87996e+012
7	7	3.56572e+012

**FOR UPDATE** ile güncelleme işlemimiz başarıyla gerçekleştirilmiş ve tüm satırlara **Random** olarak veriler eklenmiştir.

# STORED PROCEDURE'LER

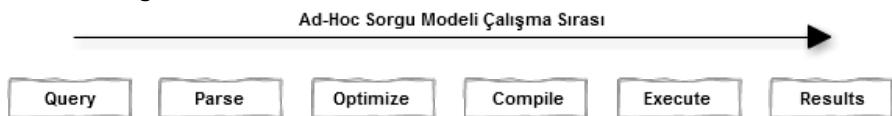
Kod bloklarının, bir kez yazılarak derlendikten sonra SQL Server'da prosedür hafızasında tutulan, tekrar çalıştırıldıklarında ise, ilk sorgu sırasında oluşturulan sorgu ve çalışma planı hazır olduğu ve hafızada tutulduğu için daha hızlı sonuç döndüren SQL Server nesneleridir. Aynı zamanda bir çok farklı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

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

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

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

## Normal sorgular;



Normal sorgularda 6 işlem aşaması vardır. Bunlar;

- **Query (Sorgu)**

Oluşturulacak SQL sorgusunun hazırlanma aşamasıdır.

- **Parse (Ayırıştırma)**

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

- **Optimize (İyileştirme)**

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

- **Compile (Derleme)**

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

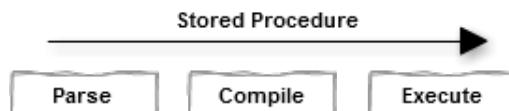
- **Execute (Çalıştırma)**

Tüm süreçlerin sonunda sorgu planı hazır olan kodların çalıştırılacağı kısımdır.

- **Results (Sonuçlar)**

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

Stored Procedure'ler;



- **Parse (Ayırıştırma)**

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

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

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

- **Compile (Derleme)**

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

- **Execute (Çalıştırma)**

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

## STORED PROCEDURE'LERİN FAYDALARI

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

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

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

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

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

## **STORED PROCEDURE TÜRLERİ**

SQL Server'da Stored Procedure'ler kullanım amaçları ve özelliklerine göre 4'ü ayrılr.

### **EXTENDED STORED PROCEDURE'LER**

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

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

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

Öncelikle **xp\_cmdshell**'i aktif edelim.

---

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

---

Sorguyu çalıştırıldıktan sonra, artık **xp\_cmdshell** kullanılabilir.



**sp\_help** ya da **sp\_helptext** kullanılarak **xp\_cmdshell** hakkında bilgi alınabilir.

**xp\_cmdshell**'i kullanarak **C:** kök dizini içerisindeki dosyaları listeleyelim.

---

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

---

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

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

---

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

---

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

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

Bir dosya içerisindeki veriyi bir başka dosyaya kopyalayalım.

`copy1` ve `copy2` adında iki not defteri dokümanı oluşturalım. Daha sonra, `copy1.txt` içeresine bazı veriler girerek aşağıdaki sorguyu çalıştırıralım.

---

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

---

output	
1	1 dosya kopyalandı.
2	NULL

Bu işlem sonucunda `copy1.txt` içerisindeki veri `copy2.txt` içeresine kopyalanacaktır.



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

## CLR STORED PROCEDURE'LER

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

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

## SİSTEM STORED PROCEDURE'LERİ

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

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

## KULLANICI TANIMLI STORED PROCEDURE'LER

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

## STORED PROCEDURE OLUSTURMAK

Ürünlerin belirli sütunlarının listesini veren bir sproc yazalım.

---

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

---

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

---

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

---

Oluşturduğumuz sproc'un kaynak koduna erişmek için;

---

```
sp_helptext 'pr_UrunleriGetir';
```

---

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

## STORED PROCEDURE İÇİN GEREKLİ İZİN VE ROLLER

Sproc oluşturabilmek için aşağıdaki izin ya da rollere ihtiyaç vardır.

Roller;

- **sysadmin**
- **db\_owner**
- **ddl\_admin**

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

## STORED PROCEDURE İÇİN KİSITLAMALAR

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

Bir sproc şu ifadeleri içeremez

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

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

## STORED PROCEDURE'Ü ÇALIŞTIRMAK

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

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

3 farklı kullanım;

- **sproc\_name;**
- **EXEC sproc\_name;**
- **EXECUTE sproc\_name;**

## NOCOUNT OTURUM PARAMETRESİNİN KULLANIMI

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

---

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

---

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

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

<b>NOCOUNT ON</b>	<b>NOCOUNT OFF</b>
Command(s) completed successfully.	(24 row(s) affected)

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

---

```
SET NOCOUNT ON
```

---

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

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

---

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

---

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

---

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

---

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

---

```
SET NOCOUNT OFF
```

---

## STORED PROCEDURE'LERDE DEĞİŞİKLİK YAPMAK

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

### Söz Dizimi:

---

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

---

Değişiklik yapmak istediğimiz sproc, önceki örneğimizdeki **pr\_UrunleriGetir** olsun. Var olan bu sproc'un kaynak kodunu görebilmek için **sp\_helpText**'i kullanalım.

---

```
sp_helpText 'pr_UrunleriGetir';
```

---

Text	
1	CREATE PROC pr_UrunlenGetir
2	AS
3	SELECT ProductID, Name, ProductNumber, ListPrice...

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

---

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

---

# STORED PROCEDURE'LARI

## YENİDEN DERLEMEK

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

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

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

---

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

---

**pr\_UrunleriGetir** isimli sproc'u her çalıştırılmasında yeniden derlenecek şekilde değiştirelim.

---

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

---

**pr\_UrunleriGetir** isimli sproc'u **WITH RECOMPILE** ile her çalıştırılmasında yeniden derlenecek şekilde düzenledik.

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

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

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

Bu işlem için;

---

```
EXECUTE prosedur_ismi WITH RECOMPILE
```

---

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

---

```
EXECUTE sp_recompile prosedur_ismi | tablo_ismi
```

---

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

## STORED PROCEDURE'LER İÇİN İZİNLERİ YÖNETMEK

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

Bir sproc'u public role kapatmak.

---

```
DENY ON prosedur_ismi TO public
```

---

Bir kullanıcıya sproc'a erişim izni vermek için;

---

```
GRANT ON prosedure_ismi TO kullanici_ismi
```

---

## STORED PROCEDURELERDE PARAMETRE KULLANIMI

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

### GİRİ PARAMETRELERİ (INPUT PARAMETERS)

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

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

---

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

---

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

## GİRİ PARAMETRELER İLE STORED PROCEDURE ÇAĞIRMAK

Dışarıdan parametre alan prosedür iki farklı şekilde kullanılabilir.

### Söz Dizimi:

---

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

---

ya da

---

```
EXEC prosedur_ismi parametre_degeri
```

---

Şimdi oluşturduğumuz `pr_UrunAra` prosedürünü çalıştıralım.

### 1. Yöntem

---

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

---

### 2. Yöntem

---

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

---

ProductID	ProductNumber	Name	ListPrice
1	522	HL Touring Seat Assembly	196,92

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

## TABLO TİPİ PARAMETRE ALAN STORED PROCEDURE'LER

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

Bu özelliği hemen uygulama üzerinde inceleyelim.

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

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

---

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

---

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

---

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

---

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

---

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

---

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

## ÇIKIŞ PARAMETRELERİLE ÇALIŞMAK (OUTPUT PARAMETERS)

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

Çıktı parametreleri için detaylı bir örnek yapalım.

Stored Procedure'den istenen özellikler:

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

Bu işlemi gerçekleştirmek için;

---

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

```

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

```

---

## ÇIKIŞ PARAMETRELERİ ALMAK

Çıkış parametrelerini alabilmek için şu yol izlenmeli.

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

Hazırladığımız uygulamanın çıkış parametre değerini alalım.

Prosedürümüz 4 parametreden oluşuyor.

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

---

-- Çıkış parametresinden alınan değerin tutulacağı değişken.

DECLARE @sonuc INT;

-- Prosedürü çalıştmak için gönderilen parametreler.

EXEC pr\_HesapMakinesi 7,6,2,@sonuc OUT;

-- Çıkış parametresinden alınan değerin gösterilmesi.

SELECT @sonuc;

---

Çıkış parametrelerini bir başka kullanımı şu şekildedir.

---

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

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

## RETURN DEYİMİ

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

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

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

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

---

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

---

Aşağıdaki şekilde çağrıabiliriz.

---

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

---

	(No column name)	(No column name)	(No column name)	(No column name)
1	Çarpım	42	Return ile Toplam :	13

## EXECUTE AS MODÜL ÇALIŞTIRMA BAĞLAMLARI

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

---

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

---

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

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

### EXECUTE AS CALLER

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

Belirtilen **ID** değerine sahip ürünü getiren bir prosedür oluşturalım.

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

---

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

---

```
WITH EXECUTE AS CALLER
```

---

## EXECUTE AS ‘KULLANICI’

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

---

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

---

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

## **EXECUTE AS SELF**

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

## **EXECUTE AS OWNER**

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

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

## **WITH RESULT SETS İLE STORED PROCEDURE ÇAĞIRMAK**

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

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

---

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

---

**WITH RESULT SETS** ile prosedürü çağıralım.

---

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

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

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

## STORED PROCEDURE GÜVENLİĞİ

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

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

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

## STORED PROCEDURE'LERİN ŞİFRELENMESİ

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

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

**pr\_UrunGetir** Stored Procedure'ünün içeriğini **sp\_helptext** sistem prosedürü ile görüntüleyelim.

---

```
sp_helptext 'pr_UrunGetir';
```

---

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

Bu kullanım ile **pr\_UrunGetir** prosedürünün içeriğini görüntüledik.

Şimdi prosedür içeriğini şifreleyelim.

---

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

---

**sp\_helptext** ile prosedür içeriğini tekrar görüntülemeye çalıştığımızda aşağıdaki sonuç ile karşılaşırız.

---

```
sp_helptext 'pr_UrunGetir';
```

---

The text for object 'pr\_UrunGetir' is encrypted.

Prosedürümüzün içeriği şifrelendi. Artık prosedürümüzün içeriği erişilemez oldu.

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

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

Ancak şifrelenmiş bir prosedürün içeriği görüntülenmeli ise;

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

## STORED PROCEDURE'LER HAKKINDA BİLGİ ALMAK

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

`Sys Procedures` kullanarak prosedürler hakkındaki bilgileri listeleyelim.

---

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

---

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

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

Seçili veritabanını değiştirmek için;

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

---

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

---

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

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

---

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

---

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

`sys.sql_modules` kullanarak prosedürler hakkında bilgileri listeleyelim.

---

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

---

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

---

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

---

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

## STORED PROCEDURE'LERİN KALDIRILMASI

Tüm nesnelerde olduğu gibi sproc silmek oldukça kolaydır.

---

```
DROP PROC|PROCEDURE sproc_ismi
```

---

Bu sorgu yapısı ile birlikte ismi belirtilen sproc ortadan kalkar.

# T-SQL İLE HATA YÖNETİMİ

12

Programcıların tecrübeli olduğu en önemli durumlardan birisi şüphesiz hata yakalama ve bu hataların doğru şekilde depolanması ile yönetilmesidir. Veritabanı gibi karmaşık yapılardan oluşan bir ortamda hataların merkezi şekilde yönetilmesi gereklidir. SQL Server, bu konuda gelişmiş özelliklere sahiptir. Kod blokları içerisinde oluşacak bir hata yönetilebileceği gibi, programcı tarafından yeni bir hata mesajı oluşturularak belirlenen durumlarda T-SQL programları içerisinde bu hata fırlatılabilir.

Aynı zamanda, bir hata yönetimi tablosunda depolanacak hata bilgilerine ait veriler, veritabanı katmanında merkezi yönetimi sağlayabileceği gibi, bu hatalar işletim sistemi loglarına kaydedilerek sistem yöneticisine de hata verilerini raporlanabilir.

## HATA MESAJLARI

Programlar içerisinde oluşan hataların düzenlenerek hatalar sonucu programların kilitlenmesi, işlemlerin gerçekleşmemesi gibi sorunların önüne geçmek için hatanın olduğu kod blokları içerisinde bu düzenlemelerin yapılması gereklidir.

Örneğin, veritabanına bir kayıt eklerken, `INT` veri tipinde beklenen kaydın `VARCHAR` veri tipinde gönderilmesi gerçekleşme olasılıkları yüksek bir hatadır. Bu duruma hazırlıklı olmak için program içerisinde ya da veri girişi sırasında koşullar oluşturularak veriler filtrelenmelidir.

Bir veri eklemeye işlemi gerçekleştirerek oluşabilecek hatayı inceleyelim.

`Production.Product` tablosundaki bir çok alan, veri girişi sırasında belirtilmesi gereken, boş geçilemez, zorunlu alanlardır. Tüm zorunlu alanları belirtmeden, sadece iki sütun değerini belirtelim.

---

```
INSERT INTO Production.Product (Name, ProductNumber)
VALUES ('Test Ürün', 'AR-5388');
```

---

Sorgu sonucunda oluşacak hata aşağıdaki gibidir.

Msg 515, Level 16, State 2, Line 2

Cannot insert the value NULL into column 'SafetyStockLevel', table 'AdventureWorks2012.Production.Product'; column does not allow nulls. INSERT fails.

SQL Server, bu hata mesajı ile `NULL` geçilemez alanların `NULL` bırakılarak bir kayıt eklemeye işlemi gerçekleştirilmek istendiğini bildirir. Hata mesajı açıklaması da buna göre yazılmıştır.

Hata mesajında, açıklama kısmından daha önemli olan kısım, Msg 515 yazılı, hata mesajının ID değerinin belirtildiği kısımdır.

Bu uyarı, `NULL` veri eklemeye ile ilgili hataların 515 hata mesaj ID değeri ile tanımladığını ve veritabanında hata mesajları tablosunda bu şekilde tutulduğunu belirtir.

Hata mesajına bakarsak, belirli bir şablon kullanıldığı aşikardır. Peki, bu hata mesajı şablonlarına nasıl ulaşabiliriz?

Örneğin; `NULL` ile ilgili işlemlerin hata kodu olan 515 no'lu hata koduna nasıl ulaşabileceğimize bakalım.

Bir sonraki **Mesajları Görüntülemek** isimli başlıkta detaylıca incelenenek olan hata mesajlarının tutulduğu `sys.messages` sistem view'inde, bize gösterilen 515 no'lu hatayı bulalım.

SQL Server içerisindeki tüm hatalar `sys.messages` view'ı içerisinde listelenir.

Bu view'i listeleyelim.

---

```
SELECT * FROM sys.messages;
```

---

	message_id	language_id	severity	is_event_logged	text
1	50001	1033	10	1	Geçerli bir ürün numarası giriniz
2	50002	1033	11	1	%d adet ürün %s kullanıcısı tarafından silindi.
3	50005	1033	16	0	Şu anki veritabanı ID değeri: %d ve veritabanı adı: ...
4	21	1033	20	0	Warning: Fatal error %d occurred at %S_DATE. No...
5	101	1033	15	0	Query not allowed in Waitfor.
6	102	1033	15	0	Incorrect syntax near "%ls".
7	103	1033	15	0	The %S_MSG that starts with "%ls" is too long. Ma...

515 no'lu hata mesajını bulalım.

---

```
SELECT * FROM sys.messages WHERE message_id = 515;
```

---

	message_id	language_id	severity	is_event_logged	text
1	515	1033	16	0	Cannot insert the value NULL into column '%ls', table '%ls'; column does ...
2	515	1031	16	0	Der Wert NULL kann in die %1!-Spalte, %2!-Tabelle nicht eingefügt werden...
3	515	1036	16	0	Impossible d'insérer la valeur NULL dans la colonne '%1!', table '%2!. Cette ...
4	515	1041	16	0	テーブル "%2!" の列 "%1!" に値 NULL を挿入できません。この列では NULL 値 ...
5	515	1030	16	0	Værdien NULL kan ikke indsættes i kolonnen "%1!", tabellen "%2!. Kolonne ...
6	515	3082	16	0	No se puede insertar el valor NULL en la columna "%1!", tabla "%2!. La colu...
7	515	1040	16	0	Impossibile inserire il valore NULL nella colonna "%1!" della tabella "%2!. La ...

22 kayıt listelendi. Bunlar incelendiğinde, her satırda verinin text sütununda farklı dillerde kayıtlar olduğu görülebilir.

İlk sırada İngilizce, sonra Almanca ve sonrasında diğer dillerle devam eden hata mesajları listeleniyor. Bu hatalardan sadece İngilizce olanını listeleyelim.

---

```
SELECT * FROM sys.messages WHERE message_id = 515 AND language_id = 1033;
```

---

	message_id	language_id	severity	is_event_logged	text
1	515	1033	16	0	Cannot insert the value NULL into column '%ls', table '%ls'; column does not allow nulls. %ls fails.

`language_id` değeri, aynı hata mesajlarının farklı dillerde açıklamalara sahip olması için kullanılır.

Bize gösterilen hata kodu, dil ayarlarımızdan dolayı İngilizce idi. Ancak aynı hatanın Türkçe açıklamasına da programlama yolu ile sahip olabiliriz.

SQL Server tarafından bize gösterilen `NULL` işlem hata mesajını, 1033 `language_id` değeriyle İngilizce açıklamalı olarak elde etmişik.

Aynı işlemin Türkçe açıklamasını 1055 `language_id` değeriyile elde edebiliriz.

```
SELECT * FROM sys.messages WHERE message_id = 515 AND language_id = 1055;
```

message_id	language_id	severity	is_event_logged	text
515	1055	16	0	NULL değeri '%2!' tablosunun '%1!' sütununa eklemez; sütun null değerlere izin vermiyor. %3! başarısız.

Hata mesajı şablonu, İngilizce ve Türkçe hata mesajlarını karşılaştıralım.

### **İngilizce hata şablonu:**

Cannot insert the value NULL into column '.\*ls', table '.\*ls'; column does not allow nulls. %ls fails.

### **İngilizce hata mesajı:**

Cannot insert the value NULL into column 'SafetyStockLevel', table 'AdventureWorks2012.Production.Product'; column does not allow nulls. INSERT fails.

### **Türkçe hata şablonu:**

NULL değeri '%2!' tablosunun '%1!' sütununa eklemez; sütun null değerlere izin vermiyor. %3! başarısız.

### **Türkçe hata mesajı:**

NULL değeri 'AdventureWorks2012.Production.Product' tablosunun 'SafetyStockLevel' sütununa eklemez; sütun null değerlere izin vermiyor. INSERT başarısız.

Bir programcının T-SQL programları geliştirirken hata mesajlarının mantığını kavraması gereklidir. Bu hata mesajlarını, sadece SQL Server kullanmaz. Bazı durumlarda, büyük projelerin hatalarını yönetmek ve hata testleri yapmak zor olabilir. Bazen bir hatayı test etmek için farklı olasılıklar hesaplamak ve hesaplanmış hatalar oluşturmak da gerekebilir. Bu tür durumlarda kendi hata yönetim katmanınızı oluşturmanız gerekecektir.

`sys.messages`'deki hata kodlarını, dil kodları ile birlikte kullanarak, gerçekleşen hatalarda, istediğiniz istemciye, istediğiniz dili kullanarak hata mesajlarını gösterebilirsiniz.

Hata şablonlarında bulunan %s, %1!, %2!, %3! ya da benzeri parametreler yer tutucu olarak kullanılır. Şablon içerisinde belirlenen bu yerlere ilgili hatanın tanımlayıcı bilgileri yerleştirilerek kullanıcıya gösterilir.

## MESAJLARI GÖRÜNTÜLEMEK

Hata yönetimi kavramına en iyi örnek SQL Server'ın kendi nesneleri ve işlemleri için kullandığı hataların açıklamalarıyla birlikte yer aldığı **sys.messages** sistem view'ıdır.

**sys.messages** sistem view'i, sistem prosedürleri, fonksiyonlar ve diğer SQL Server hatalarının açıklamaları ve dil gibi bilgileri içeren veri kaynağıdır. Bu view'deki hata mesajları SQL Server'in gelişmesiyle birlikte sürekli artacaktır. Şu an **sys.messages**'de toplam 230.000 civarında hata mesajı vardır.

Bu hatalar, çalışma ve işlemler sırasında meydana gelen hataların kullanıcıya gösterilmesi için kullanılır.

---

```
SELECT * FROM sys.messages;
```

---

message_id	language_id	severity	is_event_logged	text
1 50001	1033	10	1	Geçerli bir ürün numarası giriniz
2 50002	1033	11	1	%d adet ürün %s kullanıcısı tarafından silindi.
3 50005	1033	16	0	Şu anki veritabanı ID değeri: %d ve veritabanı adı: ...
4 21	1033	20	0	Warning: Fatal error %d occurred at %S_DATE. No...
5 101	1033	15	0	Query not allowed in Waitfor.
6 102	1033	15	0	Incorrect syntax near "%.*ls".
7 103	1033	15	0	The %S_MSG that starts with "%.*ls" is too long. Ma...

**Sys.Messages View** İçerisindeki Sütunlar:

### Message\_ID

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

### Language\_ID

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

## Severity

- **1 - 10:** Kullanıcıdan kaynaklanan bilgi içerikli hatalardır.
- **11 - 16:** Kullanıcının TRY/CATCH bloğu ile yönetebileceği hatalardır.
- **17:** Disk ya da diğer kaynakların tüketdiği durumlarda oluşur. TempDB'nin dolu olması gibi. TRY/CATCH blokları ile yakalanabilir.
- **18:** Kritik, dahili ve sistem yöneticisini ilgilendiren hatadır.
- **19:** WITH LOG özelliğinin kullanılması gereklidir. Hata NT ya da Windows Event Log'da gösterilecektir. TRY/CATCH bloğu ile yakalanabilir.
- **20 - 25:** Tehlikeli hatalardır. Kullanıcı bağlantısı sonlandırılır. WITH LOG uygulanması gereklidir. Event Log'da görüntülenebilir.

## Is\_Event\_Logged

- **0 - 1:** Bu türden bir hata oluştuğunda loglanıp loglanmayacağı belirler.

## Text

Hata mesajıdır. %s ve %d ile dışarıdan parametreler verilebilir.

# YENİ MESAJ EKLEMEK

Hata mesajlarının bulunduğu sys.messages sistem view'ine programcılar tarafından da yeni hata mesajları eklenebilir. Bu durum, SQL Server'in hata mesajlarında daha esnek olabilmesini sağlar.

SQL Server, ilk 50.000 hata mesajını kendisi belirler ve kendisi için ayırmıştır. Kullanıcılar 50.001'den itibaren yeni bir hata mesajı ekleyebilir.

Sisteme kayıtlı hata mesajları 50.000 değerine kadar olan değerleri kullanmada özel olarak SQL Server'a ayrılmıştır. Kullanılan hata mesajları sondan başa doğru listelenerek şu şekilde görülebilir.

---

```
SELECT * FROM SYS.Messages ORDER BY Message_ID DESC;
```

---

message_id	language_id	severity	is_event_logged	text
50005	1033	16	0	%u anki veritabanı ID değeri: %d ve veritabanı adı: %s.
50002	1033	11	1	%d adet ürün '%s' kullanıcısı tarafından silindi.
50001	1033	10	1	Geçerli bir ürün numarası giriniz
49913	1033	10	0	The server could not load DCOM. Software Usage Metrics cannot be started without DCOM.
49913	1031	10	0	DCOM konnte vom Server nicht geladen werden. Softwaresnutzungsmetriken können ohne...
49913	1036	10	0	Le serveur n'a pas pu charger DCOM. Impossible de démarrer les mesures d'utilisation des I...
49913	1041	10	0	サーバーは DCOM をロードできませんでした。DCOM がないと、ソフトウェアの使用状況メトリッ...

Yeni bir hata mesajı eklemek için kullanılan genel yapı aşağıdaki gibidir.

---

```
sp_addmessage @msgnum = 'mesaj_kod',
@severity = 'seviye',
@msgtext = 'mesaj',
@with_log = 'true' | 'false',
@lang = 'dil_kod',
@replace = ''
```

---

Söz dizimindeki parametrelerin bazıları şu anlama gelir.

- **@msgnum:** 50001'den itibaren bir tam sayı.
- **@lang:** Hata mesajının dil kodu. `sys.syslanguages` ya da `master.dbo.syslanguages` ile ulaşılabilir.
- **@with\_log:** True ya da False değeri alır. True ise, hata kodu `RAISERROR` ile çağrıldığında, SQL Server'in üzerinde çalıştığı işletim sisteminin event view ile görülebilen sistem log'larına mesajı ve oluş zamanının eklenmesini sağlar.
- **@replace:** Yeni bir mesaj eklemek yerine mevcut bir hatanın düzenlenmesi için kullanılır. @replace değişkeni değerine `REPLACE` olarak bildirilmesi gereklidir.

`sp_addmessage` ile yeni bir hata mesajı ekleyelim.

---

```
sp_addmessage @msgnum = '50006',
@severity = 10,
@msgtext = 'Geçerli bir ürün numarası giriniz',
@with_log = 'true';
```

---

Eklelen hata mesajını görüntüleyelim.

---

```
SELECT * FROM SYS.Messages WHERE Message_ID = 50006;
```

---

	message_id	language_id	severity	is_event_logged	text
1	50006	1033	10	1	Geçerli bir ürün numarası giriniz

`sp_addmessage` sistem prosedürünü çalıştırın uygulamanın `WITH LOG` seçeneğini kullanabilmesi için `sysAdmin` server rolüne sahip olması gereklidir.

## PARAMETRELİ HATA MESAJI TANIMLAMAK

Hata yönetimi bölümünün ilk açıklama kısmında ve sonrasında hata fırlatma ile ilgili konularda yüzeysel olarak istediğimiz parametrelî hata mesajlarını detaylandıracagız.

Hata mesajları için oluşturulan şablonlar içerisinde parametre kullanılarak, dışarıdan hata mesajının belirlenen kısımlarına değer atanması gerekebilir.

Şablonlara dışarıdan parametre atama işlemi için gerekli söz dizimi;

'hata mesajı %p1 mesaj devamı %p2 mesaj sonu', parametre1, parametre2

Hata mesajlarında kullanılan bu parametrelere yer tutucu denir. Yer tutucular için, veri tiplerine göre değişen farklı işaretler kullanılır.

% karakteri	Veri Tipi Karşılığı
d veya 1	Desimal
p	Pointer
s	String
o	Unsigned octal
u	Unsigned integer
x veya x	Unsigned hexadecimal

Yer tutucuları bulunan bir hata mesajı kaydedelim.

---

```
sp_addmessage @msgnum = 50002,
               @severity = 11,
               @msgtext = '%d adet ürün %s kullanıcısı tarafından silindi.',
               @with_log = 'true'
```

---

Bu hata mesajına, hata fırlatma teknikleri kullanılarak içerisindeki yer tutuculara dışarıdan parametre gönderilebilir. Bu işlem ile ilgili örnekleri **Hata Fırlatmak** kısmında inceleyeceğiz.

Ayrıca, yer tutucu özelliklerinin yanında flag ve genişlik bilgisi özellikleri de vardır.

- - (eksi) : Sola yanaştırma.
- + (arti) : Signed tipinin negatif mi pozitif mi olduğunu belirtir.
- 0 : Genişlik özelliğinde belirlenmiş olan genişlik değerine ulaşana kadar sayısal değerin başına sıfır konulacağını belirtir.
- # (pound) : Octal ve hex değerine bağlı olan uygun ön eki (0 ya da 0x) kullanılmasını belirtir. Sadece octal ve hex değerlere uygulanır.
- '' : Sayısal değer pozitif ise, sol kısmını boşluk (' ') ile doldurur.

Bu özelliklerin yanında, parametrenin genişlik, kısa/uzun durumu ve hassasiyeti de düzenlenlenebilir.

- **Width:** Parametre değerini tutmak için gerekli boş yer miktarını, bir integer değer ile ayarlar.
- \* (yıldız) işaretini kullanılırsa genişlik otomatik olarak belirlenir.
- **Precision:** Sayısal veri için, maksimum rakam sayısını belirler.
- **Long/Short:** Parametre integer, octal ya da hex veri tipinde ise, kısa (h) ya da uzun (l) olarak ayarlanabilir.

## HATA OLUŞTURURKEN KULLANILABILECEK ÖZELLİKLER: WITH

SQL Server, hata oluştururken kullanılabilecek bazı ek özelliklere sahiptir. Bu özellikler WITH ile birlikte kullanılır.

- LOG
- SETERROR
- NOWAIT

### WITH LOG

SQL Server'ın hata log bilgisini SQL Server log dosyasında ve **Windows Application Log** dosyasında tutmasını sağlar. Hata şiddeti 19 ve üzerinde olan hatalarda kullanılması gereklidir. Hata logları maksimum 400 bayt ile sınırlıdır.

Bu özellik için, **sysAdmin** sunucu rolüne ya da **ALTER TRACE** iznine sahip olunmalıdır.

## **WITH NOWAIT**

İstemciye hemen mesaj gönderir.

## **WITH SETERROR**

**RAISERROR** komutu çalıştığında, varsayılan olarak **RAISERROR** komutunun başarılı çalışıp çalışmadığını gösteren değeri tutar. **SETERROR** özelliği bu durumu değiştirecek, **@@ERROR** değişkeninin üretilen hata değerini tutmasını sağlar.

## **MESAJ SILMEK**

Kullanıcı tarafından tanımlanan bir mesajı silmek için, **sp\_dropmessage** sistem Stored Procedüre'ü kullanılır.

### **Söz Dizimi:**

---

```
sp_dropmessage mesaj_no
```

---

Oluşturduğumuz 50001 mesaj no'lu hata mesajını silelim.

---

```
sp_dropmessage 50001
```

---

## **OLUŞAN SON HATANIN KODUNU YAKALAMAK: @@ERROR**

Sistemde gerçekleşen bir hatanın hata kodu, **@@ERROR** hata kodu ortam fonksiyonu tarafından yakalanır. Bu fonksiyon, her işlemden değişen bir değere sahiptir. Yeni bir işleme geçtiğinde eski değeri kaybolur ve yeni değeri tutmaya başlar. Hata oluşmadığında 0 değerini tutar. Hata değerini kaybetmemek ve hafızada tutmak istendiğinde hata oluştuğu anda yeni bir kullanıcı tanımlı değişikene atanması gereklidir. Aksi halde, yeni hata değerine sahip olduğunda eskisine ulaşılamaz.

## Sıfır'a bölme hmasını @@ERROR ile yönetelim.

---

```

DECLARE @deadline INT, @hataKod INT;

SET @deadline = 0

SELECT DaysToManufacture / @deadline
FROM Production.Product
WHERE ProductID = 921

SET @hataKod = @@ERROR

IF @@ERROR <> 8134
BEGIN
PRINT CAST(@hataKod AS VARCHAR) + ' No''lu sıfır'a bölümme hatası.';
END
ELSE IF @@ERROR <> 0
BEGIN
PRINT CAST(@hataKod AS VARCHAR) + ' No''lu bilinmeyen bir hata
oluştı.';
END;

```

---

Msg 8134, Level 16, State 1, Line 5  
 Divide by zero error encountered.  
 8134 No'lu sıfır'a bölümme hatası.

Hata sonucunda **IF** koşulunda 8134 no'lu hatayı arayan koşul içeresine girdi ve hata kodu ile birlikte belirttiğimiz bilgiyi ekranda gösterdi.

Bu işlem prosedürel olarak gerçekleştirilebilir. Örneğin, hata kodu ve dil kodunu alan ve sonucunda **sys.messages** içerisindeki ilgili hata mesajı kaydını getiren bir prosedür kullanılabilir.

Aldığı parametreler ile hata göstermeyi sağlayacak prosedürü oluşturalım.

---

```

CREATE PROC pr_HataGoster(
  @hataKod INT,
  @dilKod INT
)
AS
BEGIN
  DECLARE @text VARCHAR(100);
  SELECT @text = Text FROM sys.messages
  WHERE message_id = @hataKod AND language_id = @dilKod;
  PRINT @text;
END;

```

---

Prosedürü test edelim.

---

```
EXEC pr_HataGoster 8134, 1055;
```

---

Sıfır'a bölümme hatasıyla karşılaşıldı.

- İlk parametre (8134):** Sıfır'a bölümme hmasını temsil eden hata mesaj kodu.
- İkinci parametre (1055):** Dil kodunu temsil eder. Türkçe hata mesajı dil kodu.

Yukarıda oluşturduğumuz prosedürel olmayan örneği, otomatik hale getirerek prosedür ile tekrar oluşturalım.

---

```
DECLARE @deadline INT, @hataKod INT, @dilKod INT
SET @deadline = 0
SELECT DaysToManufacture / @deadline FROM Production.Product
WHERE ProductID = 921
SET @hataKod = @@ERROR;
SET @dilKod = 1055; -- Türkçe dil kodu
IF @@ERROR <> 8134
BEGIN
    -- Sıfır'a bölümme hatasının Türkçe açıklamasını getirir.
    EXEC pr_HataGoster @hataKod, @dilKod
END
ELSE IF @@ERROR <> 0
BEGIN
    -- Hangi hata gerçekleşirse o hatanın Türkçe açıklamasını getirir.
    EXEC pr_HataGoster @hataKod, @dilKod
END;
```

---

```
Msg 8134, Level 16, State 1, Line 5
Divide by zero error encountered.
Sıfır'a bölümme hatasıyla karşılaşıldı.
```

## STORED PROCEDURE İÇERİSİNDE @@ERROR KULLANIMI

Hata yönetimi gerçekleştirilmesi gereken en gerekli program parçacıklarından biri Stored Procedure'dür. Bu programcılar içerisinde gerçekleşen işlemlerin yönetimi için `@@ERROR` kullanılabileceği gibi, `TRY/CATCH` bloğu da kullanılabilir.

`@@ERROR` ortam fonksiyonu kullanarak farklı Stored Procedure örnekleri gerçekleştirelim.

İş başvuruları gerçekleştiren çalışan adaylarının bilgilerinin tutulduğu `JobCandidate` tablosundan bir kayıt silme işlemi gerçekleştirmek için Stored Procedure geliştirelim. Ayrıca, hata yönetimini gerçekleştirmek için prosedür içerisinde `@@ERROR` fonksiyonunu kullanalım.

---

```
CREATE PROCEDURE HumanResources.pr_DeleteCandidate(
    @CanID INT
)
AS
DELETE FROM HumanResources.JobCandidate WHERE JobCandidateID = @CanID;
IF @@ERROR <> 0
BEGIN
    -- Başarısız olduğunu göstermek için 99 döndürür.
    PRINT N'Aday silme işleminde bir hata oluştu。';
    RETURN 99;
END
ELSE
BEGIN
    -- Başarılı olduğunu göstermek için 0 döndürür.
    PRINT N'İş adayı silindi。';
    RETURN 0;
END;
```

---

Prosedürü çağırarak `CandidateID` değeri 2 olan kaydı silelim.

```
(1 row(s) affected)
İş adayı silindi.
```

Prosedür başarıyla çalıştığı için herhangi bir hata vermedi ve ekranda işlemin başarılı olduğuna dair bilgi gösterdi. Prosedür, başarılı işlem yaptığı belirtmek için geriye 0 değeri döndürdü.

Ancak prosedür içerisinde bir hata olsaydı ekranda aşağıdaki hata bildirimi gösterilecekti.

Aday silme işleminde bir hata oluştu.

Prosedürün hata verdiğini bildirmek için `RETURN` ile geriye 99 değerini döndürecekti.

## HATA FIRLATMAK

Hata mesajlarına kaydedilen ya da anlık olarak gerçekleşen hataların fırlatılabilmesi için tetiklenmesi gereklidir. Bu fırlatma işlemini gerçekleştiren iki özellik vardır. Eski sürümlerden beri desteklenen **RAISERROR** ve SQL Server 2012 ile gelen yenilikler arasında olan **THROW** ifadesidir.

### **RAISERROR İFADESİ**

Hata mesajlarının devreye girebilmesi için tetiklenmesi yani hatanın fırlatılması gereklidir. Bunu gerçekleştiren ifadelerden birisi **RAISERROR** ifadesidir.

**RAISERROR** iki farklı amaç ile kullanılabilir. Bunlar;

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

#### Söz Dizimi:

---

```
RAISERROR (mesaj_kod, seviye, durum) [WITH LOG]
```

---

**RAISERROR** parametre açıklamaları;

- **mesaj\_kod**: sys.messages'de yer alan message\_id sütun değerine eşittir.
- **seviye**: Hata mesajının kritiklik seviyesini belirtir. 0-25 arasında değer alabilir.
- **durum**: Bir hata mesajı birden fazla yerde oluştuğunda, bu yerleri birbirinden ayırt etmek için kullanılır. 1- 127 arasında bir değer alabilir.
- **WITH LOG**: Oluşan hatanın loglara yazılmaması işaretlenerek tanımlanmış bile olsa loglara yazılmasını sağlar.

Anlık olarak bir hata üretelim.

---

```
RAISERROR('Mevcut bir ürünü eklemeye çalışıyorsunuz.', 10, 1);
```

---

Mevcut bir ürünü eklemeye çalışıyorsunuz.

Ekranda görüntülenen hata mesajının kırmızı olmadığını fark etmiş olmalısınız. Bunun nedeni hata seviyesinin yüksek olmamasıdır.

10 hata seviyesi çok kritik olmadığını belirtirken, aynı hata fırlatma işlemini seviye 16 ile gerçekleştirdiğimizde sonuç farklı olacaktır.

---

```
RAISERROR('Mevcut bir ürünü eklemeye çalışıyorsunuz.', 16, 1);
```

---

```
Msg 50000, Level 16, State 1, Line 1
Mevcut bir ürünü eklemeye çalışıyorsunuz.
```

Artık fırlattığımız hata daha kritik bir hata olarak algılanabilir.

Daha işlevsel ve dinamik değer üreten bir anlık hata oluşturarak fırlatalım. Bu hata mesajında, dinamik olarak değişkenlerden alınacak veritabanı ID ve veritabanı adı bilgilerini hata mesajı ile birlikte fırlatalım.

---

```
DECLARE @DBID INT;
DECLARE @DBNAME NVARCHAR(128);

SET @DBID = DB_ID();
SET @DBNAME = DB_NAME();

RAISERROR
(N'Su anki veritabanı ID değeri: %d ve veritabanı adı: %s.',
10, --Şiddet.
1, --Durum.
@DBID, --İlk argüman.
@DBNAME); --İkinci argüman.
```

---

```
Su anki veritabanı ID değeri: 7 ve veritabanı adı: AdventureWorks2012.
```

Hata mesajında %d ve %s argümanları bir yer tutucu olarak kullanılır. RAISERROR içerisinde @DBID ve @DBNAME değişken değerleri, çalışma zamanında alınarak bu yer tutucular ile gösterilecektir.

Yer tutuculara bir yenisini eklemek de kolaydır.

---

```
DECLARE @DBID INT;
DECLARE @DBNAME NVARCHAR(128);

SET @DBID = DB_ID();
SET @DBNAME = DB_NAME();

RAISERROR
(N'Su anki veritabanı ID değeri: %d ve veritabanı adı: %s. Coder : %s',
10, --Şiddet.
```

```
1, -- Durum.
@DBID, -- İlk argüman.
@DBNAME, -- İkinci argüman.
'Cihan Özhan'); -- Üçüncü argüman.
```

---

Şu anki veritabanı ID değeri: 7 ve veritabanı adı: AdventureWorks2012. Coder : Cihan Özhan

Yer tutucular ile ilgili detaylı anlatımı bu bölümdeki **Parametreli Hata Mesajları Tanımlamak** isimli kısımda bulabilirsiniz.

Yukarıdaki örneğin aynısını gerçek hata mesajı oluşturma yöntemi ile tekrar yapalım. **sys.messages** içerisinde bir hata mesajı ekleyelim. Bu hata mesajını kullanarak yeni bir hata fırlatalım.

---

```
EXECUTE sp_addmessage
      50007,
      10,
      N'Su anki veritabanı ID değeri: %d ve veritabanı adı: %s.';

DECLARE @DBID INT;
SET @DBID = DB_ID();

DECLARE @DBNAME NVARCHAR(128);
SET @DBNAME = DB_NAME();

RAISERROR (50007, 10, 1, @DBID, @DBNAME);
```

---

Şu anki veritabanı ID değeri: 7 ve veritabanı adı: AdventureWorks2012.

Eğer hata mesajının kırmızı yazı ile kritik olarak gösterilmesi isteniyorsa **RAISERROR** ikinci parametresi (seviye) 10 ve üzeri olması gereklidir.

## THROW İFADESİ

**RAISERROR** ile benzer özelliklere ve aynı amaca sahiptir. **THROW**, hata fırlatmak için kullanılır.

**THROW** ile anlık bir hata oluşturalım.

---

```
THROW 50001, 'Ürün ekleme sırasında bir hata meydana geldi.', 5;
```

---

```
Msg 50001, Level 16, State 5, Line 1
Ürün ekleme sırasında bir hata meydana geldi.
```

**THROW** ifadesini gerçek veri ve tablo üzerinde çalıştmak için örnek bir tablo oluşturalım.

---

```
USE tempdb;
GO
CREATE TABLE dbo.Deneme_Tablo
(
    sutun_1 int NOT NULL PRIMARY KEY,
    sutun_2 int NULL
);
```

---

Oluşturulan **Deneme\_Tablo** isimli tabloya veri ekleme işlemi gerçekleştirirken oluşturacak bir **PRIMARY KEY** hatasını fırlatacağız. **PRIMARY KEY** olan bir sütuna aynı değere sahip veri eklenemez. Bu durumda bir hata fırlatılması gerekecektir.

---

```
BEGIN TRY
    TRUNCATE TABLE dbo.Deneme_Tablo;
    INSERT dbo.Deneme_Tablo VALUES(1, 1);
    PRINT 'İlk Ekleme Sonrası';
    -- Msg 2627, Level 14, State 1 - PRIMARY KEY kısıtlama ihlali
    INSERT dbo.Deneme_Tablo VALUES(1, 1);
    PRINT 'İkinci Ekleme Sonrası';
END TRY
BEGIN CATCH
    PRINT 'Gerekirse burada istisna işlenebilir ve fırlatılabilir.';
    THROW;
END CATCH;
```

---

```
(1 row(s) affected)
İlk Ekleme Sonrası

(0 row(s) affected)
Gerekirse burada istisna işlenebilir ve fırlatılabilir.
Msg 2627, Level 14, State 1, Line 9
Violation of PRIMARY KEY constraint 'PK__Deneme_T__CCD8D9D094F12A42'.
Cannot insert duplicate key in object 'dbo.Deneme_Tablo'. The duplicate key value is (1).
```

---

Oluşturulan bloklar içerisinde iki farklı **INSERT** işlemi gerçekleştiriliyor. Ancak, sorgu çalışlığında sadece ilk **INSERT** işlemi başarıyla sonuçlanacaktır. İkinci **INSERT** işlemi, **sutun\_1** sütununa aynı veriyi eklemeye çalışacağı için **PRIMARY KEY** kısıtlama ihlali hatası meydana gelecek ve bu hata **THROW** ile fırlatılacaktır.

Tabloya eklenen tek kaydı listeleyelim.

---

```
SELECT * FROM Deneme_Tablo;
```

---

sutun_1	sutun_2
1	1

## HATA KONTROLÜ VE TRY-CATCH

Eski versiyonlarda, SQL Server programcılarrı içerisinde hata yönetimi gerçekleştirmek için `@@ERROR` fonksiyonu kullanılırdı. `@@ERROR` fonksiyonu bir blok içerisinde, oluşan bir hata sonucunda hatanın değerini tutar. Bir sonraki ifadeye geçtiğinde `@@ERROR` fonksiyonunun değeri değişecektir. Sonraki ifade de hata olmazsa 0, hata olursa hata kodunu tutmaya başlayacaktır. Yani, `@@ERROR` fonksiyonu sürekli farklı ifadeler için dinamik bir hata değişkenidir. Çalışan ifadede hata oluşup olmadığına bakar, oluşmadıysa 0 değeri ile sonraki ifadeye geçer ve artık önceki ifadenin değeri kaybolmuştur.

Bu nedenle, `@@ERROR` fonksiyonunun aldığı hata değerini yanında başka bir kullanıcı tanımlı değişkene atamak gereklidir.

İş bu kadar sıkıntılıyken, çözüm olarak .NET programlama dillerinde sıkça kullanılan `TRY-CATCH` bloğunun T-SQL'e eklenmesiyle daha kullanışlı bir hata yönetimi işlemi gerçekleştirilebilir oldu.

### Söz Dizimi:

---

```
BEGIN TRY
    { t-sql bloğu }
END TRY
BEGIN CATCH
    { t-sql ifadeleri }
END CATCH;
```

---

`TRY` ve `CATCH` bloklarının `BEGIN` ve `END` olarak açılış ve kapanışlarının ayrı ayrı gerçekleştigiğini görüyoruz. Eğer `TRY` bloğu kullanıldıysa `CATCH` bloğu da kullanılması gereklidir. Tek başına `TRY` bloğu tanımlanamaz.

Programlarda çok karşılaşılan hatalardan biri olan sıfır bölünme hatasını inceleyelim.

---

```

DECLARE @sayi1 INT = 5
DECLARE @sayi2 INT = 0
DECLARE @sonuc INT

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

```

---

8134 no lu hata oluştu

8134 no'lu hata sıfır bölünme hatasını temsil eder. Bu hata mesajı numarasını **sys.messages** içerisinde Türkçe açıklaması ile birlikte bulabiliriz.

---

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

---

message_id	language_id	severity	is_event_logged	text	
1	8134	1055	16	0	Sıfır bölünme hatasıyla karşılaşıldı.

Yukarıdaki 0'a bölmeye hatasının detaylarını bir Stored Procedure yardımı ile alabiliriz.

Anlık olarak hata bilgilerini getirecek bir prosedür oluşturalım.

---

```

CREATE PROCEDURE pr_HataBilgisiGetir
AS
SELECT
    ERROR_NUMBER() AS ErrorNumber,
    ERROR_SEVERITY() AS ErrorSeverity,
    ERROR_STATE() AS ErrorState,
    ERROR_PROCEDURE() AS ErrorProcedure,
    ERROR_LINE() AS ErrorLine,
    ERROR_MESSAGE() AS ErrorMessage;

```

---

Prosedürü, hata oluşacak kod bloğuna yerleştirelim.

---

```

DECLARE @sayi1 INT = 5
DECLARE @sayi2 INT = 0
DECLARE @sonuc INT

```

```
BEGIN TRY
    SET @sonuc = @sayi1 / @sayi2
END TRY
BEGIN CATCH
    EXECUTE pr_HataBilgisiGetir;
END CATCH;
```

	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	8134	16	1	NULL	6	Divide by zero error encountered.

Hata ile ilgili teknik bilgilerin listelenmesi için prosedürü kullandık.

Prosedürde yer alan fonksiyonların açıklamaları şu şekildedir.

- **ERROR\_NUMBER()** : Hatanın `sys.messages` içerisindeki hata kodu.
- **ERROR\_MESSAGE()** : Hata mesajı metni.
- **ERROR\_SEVERITY()** : Hatanın dışa dönük kritiklik durumu.
- **ERROR\_STATE()** : Hatanın sisteme dönük kritiklik seviyesi.
- **ERROR\_PROCEDURE()** : Hataya neden olan prosedür.
- **ERROR\_LINE()** : Hataya neden olan satır.

# DİNAMİK SQL NEDİR?

Veritabanı programlamada gerçekleştirilecek işlemler genel olarak daha önceden belirlenen sorunlar için oluşturulan SQL cümleleriyle gerçekleştirilir. Ancak, bazı durumlarda çözümü gereken işlemlerin çalışma anında belirlenmesi gerekebilir. Çalışma anında belirlenecek sorgular için Dinamik SQL kullanılır. Birçok büyük VTYS'de olduğu gibi Dinamik SQL özelliği SQL Server'da da desteklenmektedir.

## DİNAMİK SQL YAZMAK

SQL Server, Dinamik SQL sorguları yazabilmek için iki farklı yöntem destekler. Bu tekniklerin kendi arasında farklılıklar vardır. Performans ve kullanım olarak farklı olsa da iki teknik de Dinamik SQL sorguları yazmak için kullanılabilir.

SQL Server, dinamik sorgu işlemini gerçekleştirmek için, aldığı metinsel değeri, bir SQL sorgusu ile birleştirerek dinamik olarak yeni bir sorgu üretecek `EXEC`, `EXECUTE` fonksiyonunu ya da `SP_ExecuteSQL` Stored Procedure'ünü kullanır.

### `EXEC[UTE]`

`EXEC` fonksiyonu, Stored Procedure gibi nesnelerin çalıştırılması için de kullanılan bir fonksiyondur.

#### Söz Dizimi:

---

```
EXEC[UTE] ( { @string_degisken | [N] ' t_sql_ifadesi ' } [ + ...n ] )
```

---

`EXEC` fonksiyonu, kendi içerisinde farklı bir çalışma ortamına sahiptir. Örneğin; `EXEC` fonksiyonuna `string` olarak verilen bir SQL sorgusunun içerisinde tanımlanmış bir değişkene dışarıdan erişilemez. Aynı şekilde, fonksiyon içerisinde de dışarıdaki bir değişken çağrılamaz. Ancak değeri daha önceden alınabilir.

Dinamik SQL doğru kullanıldığında etkili bir çözüm sunabilir. `EXEC` fonksiyonu, birçok açıdan yararlı olsa da, işlemleri dinamik hale getirmek adına kullanılmaması gereklidir. Çünkü SQL Server, `EXEC` fonksiyonu içerisinde çalıştırılan sorgular için çalışma planı tutmaz. Bu da sorgular açısından performans kaybı demektir.

`EXEC` fonksiyonu, sadece yüksek gereklilik duyulan dinamik sorgular için kullanılmalıdır.

`EXEC` fonksiyonu için örnek uygulamalar oluşturarak, fonksiyonun hangi amaçlar ile kullanılabileceğini inceleyelim.

Herhangi bir basit veri seçme sorgusu dahil `EXEC` ile kullanılabilir.

```
EXEC ('SELECT * FROM Production.Product');
```

	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

`String` ile bir değişkenin `EXEC` fonksiyonuna atanması ile kullanılabilir.

```
DECLARE @TabloAd SYSNAME ='Sales.SalesOrderHeader';
EXECUTE ('SELECT * FROM ' + @TabloAd);
```

	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	SalesOrderNumber
1	43659	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043659
2	43660	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043660
3	43661	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043661
4	43662	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043662
5	43663	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043663
6	43664	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043664
7	43665	3	2005-07-01 00:00:00,000	2005-07-13 00:00:00,000	2005-07-08 00:00:00,000	5	0	S043665

**EXEC** öncesinde bir değişkene atanmış SQL sorgusunun fonksiyona atanması ile de kullanılabilir.

---

```
DECLARE @SQL VARCHAR(256);
SET @SQL = 'SELECT * FROM Production.Product';
EXEC (@SQL);
```

	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

**EXEC** fonksiyonu içerisinde oluşturulacak bir sorguda kullanılan sütun için takma ismi dışarıdan alarak dinamik bir sorgu hazırlayalım.

---

```
DECLARE @TakmaAd VARCHAR(6) = 'ÜrünAd';
EXEC ('SELECT Name AS ' + @TakmaAd + ' FROM Production.Product');
```

	ÜrünAd
1	Adjustable Race
2	All-Purpose Bike Stand
3	AWC Logo Cap
4	BB Ball Bearing
5	Bearing Ball
6	Bike Wash - Dissolver
7	Blade

Şema adı, tablo adı, sütun adı, karşılaştırma operatörü ve karşılaştırılacak değeri, çalışma zamanında alalım ve sonucu listeleyelim.

---

```
DECLARE @table VARCHAR(128);
DECLARE @schema VARCHAR(128);
DECLARE @column VARCHAR(128);
DECLARE @exp VARCHAR(4);
DECLARE @value VARCHAR(128);

SET @schema = 'Production'
SET @table = 'Product'
SET @column = 'ProductID'
SET @exp = '='
SET @value = '1'
```

```
EXEC('SELECT * FROM ' + @schema + '.' + @table + ' WHERE '
      +@column
      +@exp
      +@value);
```

---

	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

Bu sorgu ile isteğimiz tam olarak şu idi:

---

```
SELECT * FROM Production.Product WHERE ProductID = 1
```

---

	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

Tabi ki isterseniz, sorguda şema, tablo, sütun, operatör ve değer olmak üzere 5 parametreyi de değiştirerek farklı tablolardan farklı koşullarda istekler oluşturabilirsiniz.

Aynı soruyu aşağıdaki SET işlemlerini gerçekleştirerek çalışıtmayı deneyin.

---

```
SET @schema = 'Person'
SET @table = 'Person'
SET @column = 'BusinessEntityID'
SET @exp = '<='
SET @value = '7'
```

---

Dinamik SQL ile bir tablo üzerinde dinamik sorgu oluşturalım.

İlk olarak tablomuzu oluşturalım.

---

```
CREATE TABLE DynamicSQL
(
    TableID INT IDENTITY NOT NULL CONSTRAINT PKDynSQL PRIMARY KEY,
    SchemaName VARCHAR(150),
    TableName VARCHAR(150),
    Create_Date SMALLDATETIME
);
```

---

**SELECT** ile oluşturduğumuz tabloya kayıt ekleyelim.

---

```
INSERT INTO DynamicSQL
SELECT S.Name AS SchemaName, T.Name AS TableName, T.Create_date AS OTarih
FROM Sys.Schemas S
JOIN Sys.Tables T
ON S.Schema_ID = T.Schema_ID;
```

---

Kayıtları eklediğimiz tabloyu listeleyelim.

---

```
SELECT * FROM DynamicSQL;
```

---

TableID	SchemaName	TableName	Create_Date
1	1	Production	ScrapReason
2	2	HumanResources	Shift
3	3	Production	ProductCategory
4	4	Purchasing	ShipMethod
5	5	Production	ProductCostHistory
6	6	Production	ProductDescription
7	7	Sales	ShoppingCartItem

Artık dinamik SQL sorgumuzu hazırlayabiliriz.

---

```
DECLARE @SchemaName VARCHAR(128);
DECLARE @TableName VARCHAR(128);

SELECT @SchemaName = SchemaName, @TableName = TableName
FROM DynamicSQL WHERE TableID = 7;

EXEC ('SELECT * FROM ' + @SchemaName + '.' + @TableName);
```

---

	ShoppingCartItemID	ShoppingCartID	Quantity	ProductID	DateCreated	ModifiedDate
1	2	14951	3	862	2007-12-11 17:54:07.603	2007-12-11 17:54:07.603
2	4	20621	4	881	2007-12-11 17:54:07.603	2007-12-11 17:54:07.603
3	5	20621	7	874	2007-12-11 17:54:07.603	2007-12-11 17:54:07.603

Bu dinamik sorgu ile **DynamicSQL** tablosundan, istediğiniz tablonun **TableID** değerini sorgulayabilirsiniz.

## **EXEC İÇERİSİNDE FONKSİYONLAR KULLANILABİLİR Mİ?**

`EXEC` mimarisi gereği, içerisindeki verinin daha önceden çözümlenmesi gerekir. Yani direkt olarak `EXEC` içerisinde bir fonksiyon kullanılamaz.

`EXEC` içerisinde bir fonksiyon kullanılması gerekiyor ise; `EXEC` işleminden önce, fonksiyon ile ilgili işlemler yapılır ve bu değerler bir değişkene atanır. Daha sonra içerisindeki veriyi işlemek için, `EXEC` komutuna verilen değişken, herhangi bir hataya sebebiyet vermeyecektir. Ancak `EXEC` içerisinde bir fonksiyon kullanılırsa, bu durum hataya yol açar.

## **EXEC İLE STORED PROCEDURE KULLANIMI**

Stored Procedure içerisinde dinamik sorgular oluşturulabilir. Bu yöntem performans olarak önerilmese de teknik olarak kullanılabilirdir.

Dışarıdan parametre olarak Stored Procedure ismini alarak prosedür çağrıran bir Sproc oluşturalım.

---

```
CREATE PROC pr_ProcedureCall(
    @sp_ad  VARCHAR(2000)
)
AS
EXEC (@sp_ad);
```

---

Kilitler ile ilgili bir sistem prosedürü olan `sp_lock` prosedürünü çağıralım.

---

```
pr_ProcedureCall 'sp_lock';
```

---

ya da

---



---

```
EXEC pr_ProcedureCall 'sp_lock';
```

---

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	51	5	0	0	DB		S	GRANT
2	52	7	0	0	MD	1(22:3:0)	Sch-S	GRANT
3	52	7	0	0	MD	1(22:2:0)	Sch-S	GRANT
4	52	7	0	0	MD	1(22:1:0)	Sch-S	GRANT
5	52	7	0	0	MD	1(22:4:0)	Sch-S	GRANT
6	52	32767	0	0	MD	1(f72ef125:1:0)	Sch-S	GRANT
7	52	32767	0	0	MD	1(29:13:0)	Sch-S	GRANT

## DİNAMİK SQL GÜVENLİK SORUNSALI

Bir dinamik sorgu oluştururken, diğer tüm sorgulara göre daha dikkatli davranış gereklidir. Gerekli güvenlik önlemleri alınmazsa, dinamik sorgular bir para kasasına açılan küçük bir deliğe benzer. Hırsızın birisi, para kasasındaki bu deliği bir gün fark ederek tüm varlığını ele geçirebilir. Dinamik SQL ile oluşturulan sorgu da, eğer sorgu içerisinde bir filtreleme ve gerekli güvenlik koşulları oluşturulmazsa, kötü niyetli kişiler tarafından hiç tahmin etmeyeceğiniz ve istemeyeceğiniz sorgular çalıştırılarak, veri kaybına kadar gidebilecek sorunlar yaşatabilir.

Az önce geliştirdiğimiz ve basit olarak, sadece bir prosedür ismi vererek bir prosedürün çağırmasını sağladığımız sorguyu güvenlik konusu ile tekrar ele alalım.

---

```
pr_ProcedureCall 'sp_lock';
```

---

Yukarıdaki sorgu sadece `sp_lock` prosedürüne çağrıracaktır. Peki, kötü niyetli birinin farklı bir sorgu çalıştırması engelleyen bir önlem alındı mı?

`pr_ProcedureCall` prosedürünü `AdventureWorks` veritabanında oluşturmuştık. Prosedürde herhangi bir değişiklik yapmadan aşağıdaki şekilde kullanalım.

---

```
pr_ProcedureCall 'USE DIJIBIL; DROP TABLE Makaleler';
```

---

`AdventureWorks` veritabanını kullanması gereken bir prosedüre `USE DIJIBIL` diyerek `DIJIBIL` veritabanını kullanma isteği gönderebildik. Ayriyeten, noktalı virgülden sonra ikinci sorguyu da çalıştırarak `DIJIBIL` veritabanındaki `Makaleler` tablosunu silme isteğini de rahatlıkla gönderebildik.

Bu sorgu sonucunda, basit bir prosedürdeki güvenlik hatası ile farklı bir veritabanındaki bir tabloyu silebildik.

Bu tür dışarıdan parametre alarak çalışan tüm sorgularda, sadece sizin belirlediğiniz kriterlerde işlem yapabilecek yetki ve esneklik dışında hiç bir işleme müsaade edilmemelidir.

Ürün isimleri arasında arama yapacak bir Stored Procedure geliştirelim. Bu prosedür içerisindeki sorguları **EXEC** fonksiyonu ile çalışıralım.

---

```
CREATE PROCEDURE sp_ProductDynamicSP(
    @val VARCHAR(10)
)
AS
EXEC ('SELECT Name, ProductNumber
    FROM Production.Product
    WHERE Name LIKE ''%'' + @val + '%'''');
```

---

Prosedürü çağırıralım.

---

```
EXEC sp_ProductDynamicSP 'jus'
```

---

	Name	ProductNumber
1	Adjustable Race	AR-5381

---

```
EXEC sp_ProductDynamicSP 'a'
```

---

	Name	ProductNumber
1	Adjustable Race	AR-5381
2	Bearing Ball	BA-8327
3	BB Ball Bearing	BE-2349
4	Headset Ball Bearings	BE-2908
5	Blade	BL-2036
6	LL Crankarm	CA-5965
7	ML Crankarm	CA-6738

---

Dışarıdan bir ürün **ProductID** değerini parametre alarak sonucunda ürünün **Name** ve **ProductNumber** bilgilerini getiren prosedür oluşturalım.

---

```
CREATE PROCEDURE sp_GetProductByIDdynSP (
    @productID VARCHAR(10)
)
AS
EXEC ('SELECT Name, ProductNumber
    FROM Production.Product
    WHERE ProductID = ' + @productID + '');
```

---

Prosedüre parametre değeri vererek çağıralım.

---

```
EXEC sp_GetProductByIDdynSP 1;
```

---

	Name	ProductNumber
1	Adjustable Race	AR-5381

## EXEC FONKSİYONU İÇERİSİNDE TÜR DÖNÜŞÜMÜ

T-SQL sorgulamalarında veri tipi dönüştürme işlemleri gerçekleştirmek mümkündür. Bu işlem için sınırlı sayıda kısıtlama olsa da filtrelemelerde veri tipi dönüşümü desteklenir. Ancak dinamik sorgu üretirken, **EXEC** fonksiyonunda, fonksiyon içerisinde tür dönüşümü yapılamaz.

Yukarıdaki prosedürü dışarıdan **INT** veri tipinde değer alacak şekilde düzenleyelim.

---

```
CREATE PROCEDURE sp_GetProductByIDdynSP( @productID INT )
AS
EXEC('SELECT Name, ProductNumber
      FROM Production.Product
      WHERE ProductID = ' + CONVERT(VARCHAR(5), @productID));
```

---

T-SQL standartlarına göre herhangi bir sorun yok. Ancak dinamik sorgulama fonksiyonu **EXEC** için bu bir sorundur. Yukarıdaki dinamik sorgu kullanımı hata üretecektir.

Tür dönüşümü işlemini **EXEC** fonksiyonunun dışında gerçekleştirmek gereklidir.

Aynı işlemi gerçekleştiren aşağıdaki sorgu başarıyla çalışacaktır.

---

```
CREATE PROCEDURE sp_GetProductByIDdynSP( @productID INT )
AS
DECLARE @val VARCHAR(5) = CONVERT(VARCHAR(5), @productID);
EXEC('SELECT Name, ProductNumber
      FROM Production.Product
      WHERE ProductID = ' + @val);
```

---

Prosedürü tekrar çağıralım.

---

```
EXEC sp_GetProductByIDdynSP 1;
```

---

Name	ProductNumber
1	Adjustable Race
	AR-5381

**EXEC** fonksiyonunu kullanarak geçerli veritabanları üzerine farklı bir örnek oluşturalım.

---

```
USE AdventureWorks2012
GO
DECLARE @cmd VARCHAR(4000);
SET @cmd = 'EXEC spCurrDB';
SET @cmd = 'SELECT ''Geçerli Veritabanı: ['''+D.NAME+'''']'''
+ ' FROM master..sysdatabases d, master..sysprocesses p '
+ ' WHERE p.spid = @@SPID and p.dbid = d.dbid ';
EXEC (@cmd);
EXEC (N'USE master;'+@cmd);
EXEC (@cmd);
```

---

(No column name)
1 Geçerli Veritabanı: [AdventureWorks2012]
(No column name)
1 Geçerli Veritabanı: [master]
(No column name)
1 Geçerli Veritabanı: [AdventureWorks2012]

Bu örnekte, ilk olarak geçerli veritabanı gösteriliyor. Daha sonra master veritabanı seçilerek geçerli veritabanı tekrar gösteriliyor ve son olarak ilk geçerli veritabanına dönülerek gösteriliyor.

## SP\_EXECUTE SQL İLE DİNAMİK SORGU ÇALIŞTIRMAK

**EXEC** fonksiyonuna göre bazı durumlarda daha yüksek performanslı bir sistem prosedürüdür. Bu prosedür ile **EXEC** fonksiyonunun aksine, sorgular için bir çalışma planı oluşturulur. Bu nedenle, sonraki sorgulama işleminde daha performanslı bir sonuç elde edilir.

**sp\_executesql** prosedürü dışarıdan parametre alabilir. Bu da sorgularda daha esnek ve avantajlı bir kullanım sağlayabilir.

Basit bir **SELECT** cümlesi ile kullanımı aşağıdaki gibidir.

---

```
EXECUTE sp_executesql N'SELECT * FROM Purchasing.PurchaseOrderHeader';
```

---

	PurchaseOrderID	RevisionNumber	Status	EmployeeID	VendorID	ShipMethodID	OrderDate	ShipDate	SubTotal
1	1	1	4	258	1580	3	2005-05-17 00:00:00.000	2005-05-26 00:00:00.000	201,04
2	2	1	1	254	1496	5	2005-05-17 00:00:00.000	2005-05-26 00:00:00.000	272,1015
3	3	1	4	257	1494	2	2005-05-17 00:00:00.000	2005-05-26 00:00:00.000	8847,30
4	4	1	3	261	1650	5	2005-05-17 00:00:00.000	2005-05-26 00:00:00.000	171,0765
5	5	1	4	251	1654	4	2005-05-31 00:00:00.000	2005-06-09 00:00:00.000	20397,30
6	6	1	4	253	1664	3	2005-05-31 00:00:00.000	2005-06-09 00:00:00.000	14628,05
7	7	1	4	255	1678	3	2005-05-31 00:00:00.000	2005-06-09 00:00:00.000	58685,55

Tam nesne ismi belirtilmesi gerekiğinde üç parçalı isimlendirme kuralı kullanılabilir. Bu sistem prosedürü ile üç parçalı isimlendirme şu şekilde yapılabilir.

---

```
EXEC AdventureWorks2012.dbo.sp_executesql N'EXEC sp_help';
```

---

**SP\_ExecuteSQL** prosedürü ile aşağıdaki gibi girdi ve çıktı parametreleri de kullanılabilir.

```
DECLARE @SQL NVARCHAR(MAX),
        @ParmDefinition NVARCHAR(1024)
DECLARE @ListPrice MONEY = 2000.0,
        @LastProduct VARCHAR(64)
SET @SQL = N'SELECT @pLastProduct = MAX(Name)
              FROM AdventureWorks2012.Production.Product
              WHERE ListPrice >= @pListPrice'
SET @ParmDefinition = N'@pListPrice MONEY,
                      @pLastProduct VARCHAR(64) OUTPUT'
EXECUTE sp_executesql @SQL, @ParmDefinition, @pListPrice = @ListPrice,
                      @pLastProduct = @LastProduct OUTPUT
SELECT [ListPrice >=] = @ListPrice, LastProduct = @LastProduct;
```

---

	List Price >=	Last Product
1	2000,00	Touring-1000 Yellow, 60

Dinamik SQL ile tüm veritabanlarındaki tablo sayısını hesaplayalım.

```

DECLARE @SQL NVARCHAR(MAX), @dbName SYSNAME;
DECLARE DBcursor CURSOR FOR
    SELECT NAME FROM master.dbo.sysdatabases
    WHERE NAME NOT IN ('master','tempdb','model','msdb')
        AND DATABASEPROPERTYEX(NAME,'status') = 'ONLINE' ORDER BY NAME;
OPEN DBcursor; FETCH DBcursor INTO @dbName;
WHILE (@@FETCH_STATUS = 0)
BEGIN
    DECLARE @dbContext NVARCHAR(256) = @dbName+'.dbo.'+sp_executesql'
    SET @SQL = 'SELECT ''Database: ' + @dbName +
               ' TABLE COUNT'' = COUNT(*) FROM sys.tables';
    PRINT @SQL;
    EXEC @dbContext @SQL;
    FETCH DBcursor INTO @dbName;
END;
CLOSE DBcursor; DEALLOCATE DBcursor;

```

Database: AdventureWorks2012 TABLE COUNT	
1	78
Database: DIJIBIL TABLE COUNT	
1	7
Database: DijiLabs TABLE COUNT	
1	2
Database: ReportServer TABLE COUNT	
1	34
Database: ReportServerTempDB TABLE COUNT	
1	13

## DİNAMİK SQL İLE SIRALAMA İŞLEMİ

Bir dinamik sorgu içerisinde `ORDER BY` ile sıralama özelliği ekleyerek dinamik sıralama gerçekleştirebiliriz.

```

DECLARE @SQL NVARCHAR(MAX) = 'SELECT ProductID, Name, ListPrice, Color
                             FROM Production.Product ORDER BY Name '
DECLARE @Collation NVARCHAR(MAX);
SET @Collation = 'COLLATE SQL_Latin1_General_CI_AS'
SET @SQL = @SQL + @Collation

```

```
PRINT @SQL
EXEC sp_executesql @SQL;
```

---

	ProductID	Name	ListPrice	Color
1	1	Adjustable Race	0,00	NULL
2	879	All-Purpose Bike Stand	159,00	NULL
3	712	AWC Logo Cap	8,99	Multi
4	3	BB Ball Bearing	0,00	NULL
5	2	Bearing Ball	0,00	NULL
6	877	Bike Wash - Dissolver	7,95	NULL
7	316	Blade	0,00	NULL

## SP\_EXECUTE SQL İLE STORED PROCEDURE KULLANIMI

`SP_ExecuteSQL` ile de Sproc kullanılabilir. Bu işlemin `EXEC` fonksiyonu ile sproc oluşturmaktan bir farkı yoktur.

`SP_ExecuteSQL` ile ürün araması gerçekleştiren bir Sproc oluşturalım.

---

```
CREATE PROCEDURE pr_UrunAra @ProductName VARCHAR(32) = NULL
AS
BEGIN
    DECLARE @SQL NVARCHAR(MAX)
    SELECT @SQL = ' SELECT ProductID, ProductName = Name,
                    Color, ListPrice ' + CHAR(10) +
                  ' FROM Production.Product' + CHAR(10) +
                  ' WHERE 1 = 1 ' + CHAR(10)
    IF @ProductName IS NOT NULL
        SELECT @SQL = @SQL + ' AND Name LIKE @pProductName'
    PRINT @SQL
    EXEC sp_executesql @SQL, N'@pProductName VARCHAR(32)', @ProductName
END
GO
```

---

Prosedürü çağıralım.

---

```
EXEC pr_UrunAra '%bike%';
```

---

	ProductID	ProductName	Color	ListPrice
1	879	All-Purpose Bike Stand	NULL	159,00
2	877	Bike Wash - Dissolver	NULL	7,95
3	876	Hitch Rack - 4-Bike	NULL	120,00
4	710	Mountain Bike Socks, L	White	9,50
5	709	Mountain Bike Socks, M	White	9,50

## SP\_EXECUTE SQL İLE INSERT İŞLEMİ

Dinamik sorgudan dönen veriyi gerçek ya da geçici bir tabloda saklayarak, veri üzerinde yeni bir sorgu oluşturulabilir.

Geçici bir tablo oluşturalım ve dinamik sorgudan dönen değeri bu geçici tabloya ekleyelim.

---

```
CREATE TABLE #Product (ProductID int, ProductName varchar(64));
INSERT #Product
EXEC sp_executesql N'SELECT ProductID, Name FROM Production.Product';
SELECT * FROM #Product ORDER BY ProductName;
GO
DROP TABLE #Product;
```

---

	ProductID	ProductName
1	1	Adjustable Race
2	879	All-Purpose Bike Stand
3	712	AWC Logo Cap
4	3	BB Ball Bearing
5	2	Bearing Ball
6	877	Bike Wash - Dissolver
7	316	Blade

## SP\_EXECUTE SQL İLE VERİTABANI OLUŞTURMAK

Dinamik sorguların sık kullanıldığı alanlardan biri de toplu program parçalarından oluşan sorgu script'leridir. Örneğin; sütunlarını sizin belirlediğiniz bir tablo oluşturmak ya da bir veritabanı oluşturmak için kullanılabilir.

Dinamik olarak basit bir veritabanı oluşturmak için **SP\_ExecuteSQL** kullanalım.

---

```
CREATE PROC pr_CreateDB @DBName SYSNAME
AS
BEGIN
```

```
DECLARE @SQL NVARCHAR(255) = 'CREATE DATABASE ' + @DBName;
EXEC sp_executeSQL @SQL;
END;
```

---

Prosedürü çalıştırarak örnek bir veritabanı oluşturalım.

---

```
EXEC pr_CreateDB 'ornek_db';
```

---



# KULLANICI TANIMLI FONKSIYONLAR

14

Kullanıcı Tanımlı Fonksiyonlar (*User Defined Functions*), sorgu tekrarlarını önlemek amacı ile iş parçacıkları oluşturmak için kullanılır.

Kullanıcı tanımlı fonksiyonlar, dışarıdan parametre alabilir, **IF ELSE** gibi akış kontrol ifadeleri içerebilirler. Parse edilir, derlenir ve tampon hafızadan çağrılabılır. Stored Procedure ve view nesnelerine benzerler. Bir view gibi **SELECT** sorgularında kullanılabilir. Bir view ile parametrelî işlem yapamazsınız, ancak bu KTF ile mümkün değildir. Bir Stored Procedure'den alınan sonucu **SELECT** sorgunuzda etkin olarak kullanamazsınız, ancak KTF ile mümkün değildir.

Kullanıcı tanımlı fonksiyonlar, view'lerin esneklik ve kullanılabilirliği ile Stored Procedure'lerin parametre kullanabilme, tampon hafızadan çağrılabİLME, parse edilme, derlenme gibi bir çok mimari yeteneklerinin birleşimi olarak düşünülebilir.

Bir KTF, veritabanında veri seçme işlemlerini gerçekleştirmek üzere iş parçacığı olarak geliştirilir. KTF çalışmasını tamamladıktan sonra, kayıtlar üzerinde herhangi bir değişiklik (side-effect / yan etki) yapmamış olması gereklidir.

Kullanıcı Tanımlı Fonksiyonlar ile neler yapılabilir?

- Sürekli gerçekleştirilen işlemler fonksiyonel hale getirilebilir.
- SQL Server fonksiyonel hale getirilebilir. SQL Server tarafından desteklenmeyen bir fonksiyon geliştirilebilir. Örneğin, doğum tarihi ve şu anki tarihi vererek yaş hesaplatma işlemi bir fonksiyon geliştirerek yapılabilir.

Ya da kendi algoritmanıza göre bir metin şifreleme işlemi gerçekleştirmek için fonksiyon geliştirebilirsiniz. Hesap makinesi gibi işlemler yapan bir fonksiyon ya da `PI()` sayısı ile işlem yapmaya yarayan bir fonksiyon iyi birer örnek olabilir.

- Bu nesneler T-SQL ile geliştirilebileceği gibi, CLR ile de geliştirilebilirler.

## KULLANICI TANIMLI FONKSİYON ÇEŞİTLERİ

KTF nesneleri fonksiyoneldir. Bir KTF, geri dönüş tipi olarak `INT`, `VARCHAR`, `DATETIME` gibi skaler veri tiplerini döndürebileceği gibi, bir tablo tipli değişken de döndürebilir.

KTF, `SELECT` işlemleri için geniş ve işlevsel yeteneklere sahiptir. KTF nesneleri iki ana başlıkta inceleyeceğiz.

- Skaler Kullanıcı Tanımlı Fonksiyonlar
- Tablo Kullanıcı Tanımlı Fonksiyonlar

### SKALER KULLANICI TANIMLI FONKSİYONLAR

Skaler fonksiyon, bir tek değer döndüren fonksiyondur. Birden fazla parametre alabilir, ancak sonuç olarak tek bir değer döndürür. SQL Server içerisindeki kullanılan `GETDATE()` bir sistem fonksiyonudur. `GETDATE()` fonksiyonu, o anki sistem zaman bilgisini alarak geriye sadece bu bilgiyi döndürür. Bu özellik sistem tarafından sistem fonksiyonlarında kullanılabileceği gibi, T-SQL geliştiricileri tarafından da bu tür örnek fonksiyonlar geliştirilebilir.

PI sayısı en kısa haliyle 3,14 olarak kabul edilir. PI sayısı ile işlem yapılması gerekiğinde, tanımlayacağınız bir `PI()` fonksiyonu içerisinde gerekli parametreleri göndererek, fonksiyon içerisinde 3,14 değerini kullanarak işlem yapar ve sonucu tek değer olarak döndürebilirsiniz.

Geliştirilen uygulamada PI sayısı kullanmak isteniyor olabilir. Bize PI sayısını döndürecek bir fonksiyon geliştirelim.

---

```
CREATE FUNCTION PINedir()
RETURNS NUMERIC(5,2)
AS
BEGIN
    RETURN 3.14;
END;
```

---

KTF, numeric geri dönüş tipine sahip ve herhangi bir parametre almadan geriye 3.14 değerini döndürecektr. KTF sonucunu, başka bir sorgu içerisinde kullanabileceğiniz gibi tek olarak da kullanabilirsiniz.

`PInedir()` fonksiyonunu tek olarak çağrırmak için;

```
SELECT dbo.PInedir();
```

(No column name)

1 3.14

ya da

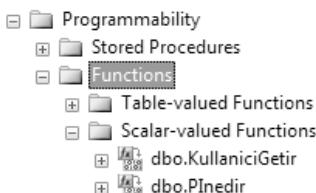
```
PRINT dbo.PInedir();
```

Messages

3.14

İki yöntemle de çağrılabilen fonksiyonun adından önce kullanılan şema adı (`dbo`) dikkatinizi çekmiş olmalıdır. Şema adı olmadan çalıştırıldığınızda sorgunuz hata üretecektir. KTF kullanırken şema adını yazmalısınız.

`PInedir()` fonksiyonunu SSMS aracından da görebilir ve yönetebiliriz.



PI sayısına ihtiyacınız olduğunda, sistem fonksiyonları içerisindeki `PI()` fonksiyonunu kullanabilirsiniz.

```
SELECT PI();
```

(No column name)

1 3.14159265358979

KTF, genel olarak veritabanında bir tablo üzerinden sorgu gerçekleştirmek için kullanılır. Tablo kullanarak bir işlem gerçekleştirelim.

Ürünler tablosunda ne kadar ürün olduğunu hesaplayalım.

```
CREATE FUNCTION dbo.UrunToplamSayi()
RETURNS INT
AS
BEGIN
    DECLARE @toplam INT;
    SELECT @toplam = COUNT(ProductID) FROM Production.Product;
    RETURN @toplam;
END;
```

Oluşturulan KTF nesnesini çağırıralım.

---

```
SELECT dbo.UrunToplamSayi();
```

---

	(No column name)
1	505

KTF, herhangi bir parametre almadan da çalışabilir. Ancak genel kullanım ve en güçlü olduğu alan tabi ki parametreli kullanımıdır.

Kullanıcının **BusinessEntityID** değerini alarak, ad ve soyad bilgilerini birleştirip geri döndürelim.

---

```
CREATE FUNCTION dbo.KullaniciGetir(@KullniciKod INT = NULL)
RETURNS VARCHAR(100)
AS
BEGIN
    DECLARE @ad_soyad VARCHAR(100)
    SELECT @ad_soyad = FirstName + ' ' + LastName
    FROM Person.Person WHERE BusinessEntityID = @KullniciKod
    RETURN @ad_soyad
END;
```

---

**BusinessEntityID** değeri 1 olan kaydı listeleyelim.

---

```
SELECT dbo.KullaniciGetir(1);
```

---

	(No column name)
1	Ken Sánchez

Fonksiyonu inceleyelim,

**KullaniciGetir()** fonksiyonuna dışarıdan **INT** veri tipinde bir **BusinessEntityID** değeri alıyoruz. Fonksiyon içerisindeki işlemler sonucunda kullanıcının ad ve soyad bilgilerini birleştirerek geri döndüreceğiz. Bu nedenle, fonksiyonun geri dönüş tipi olarak **VARCHAR(100)** belirledik. Geri dönüş tipini belirtmek için **RETURNS** komutunu kullandık. Daha sonra işlemlerimizi gerçekleştirmek için **BEGIN ... END** blokları arasında değişken tanımlama ve **SELECT** sorgumuzu yazıyoruz. Bu sorgu sonucunda dönen değeri **@ad\_soyad** değişkenine atayarak **RETURN** ile fonksiyon dışına gönderiyoruz.

## TÜRETİLMİŞ SÜTUN OLARAK SKALER FONKSİYON

Skaler fonksiyonlar tablolarda türetilmiş sütun olarak kullanılabilir.

Daha önce oluşturduğumuz **dbo.KullaniciGetir()** fonksiyonunu, bir **SELECT** sorgusu içerisinde türetilmiş sütun olarak kullanalım.

---

```
SELECT
    BusinessEntityID, PersonType, Title,
    dbo.KullaniciGetir(BusinessEntityID) AS AdSoyad
FROM Person.Person;
```

---

	BusinessEntityID	PersonType	Title	AdSoyad
1	1	EM	NULL	Ken Sánchez
2	2	EM	NULL	Teri Duffy
3	3	EM	NULL	Roberto Tamburello
4	4	EM	NULL	Rob Walters
5	5	EM	Ms.	Gail Erickson
6	6	EM	Mr.	Jossef Goldberg
7	7	EM	NULL	Dylan Miller

Fonksiyonun `Person.Person` tablosu içerisinde bir sütun olarak yer almasını istersek;

---

```
ALTER TABLE Person.Person
ADD AdSoyad AS dbo.KullaniciGetir(BusinessEntityID);
```

---

## TABLO DÖNDÜREN

### KULLANICI TANIMLI FONKSİYONLAR

Tablo değeri döndüren fonksiyonların skaler fonksiyonlardan farkı, geriye tek bir değer değil, tablo tipinde değer döndürmesidir.

Bu fonksiyonlar parametre alabildiği gibi, içerisinde bir tablo yapısı oluşturularak, oluşturulan bu tabloyu `RETURN` ile geri döndürebilir.

Tablo döndüren fonksiyonlar kendi içerisinde ikiye ayrılır.

- Satırda Tablo Döndüren Fonksiyonlar (*Inline Table-Valued Functions*)
- Çoklu İfade İle Tablo Döndüren Fonksiyonlar (*Multi-Statement Table-Valued Functions*)

### SATIRDAN TABLO DÖNDÜREN FONKSİYONLAR

Tablo döndüren fonksiyonlar, view gibi sorgulanabilen, Stored Procedure gibi parametre alan KTF nesneleridir.

Basit bir ürün arama fonksiyonu geliştirelim.

---

```
CREATE FUNCTION fnc_UrunAra(
    @ara VARCHAR(10)
) RETURNS TABLE
AS
BEGIN
    RETURN SELECT * FROM Production.Product
    WHERE Name LIKE '%' + @ara + '%';
END
```

---

Fonksiyonu parametre ile çağıralım.

---

```
SELECT * FROM dbo.fnc_UrunAra('Be');
```

---

	ProductID	Name	Product Number	MakeFlag	FinishedGoodsFlag	Color	SafetyStockLevel	ReorderPoint	StandardCost	ListPrice
1	3	BB Ball Bearing	BE-2349	1	0	NULL	800	600	0.00	0.00
2	2	Bearing Ball	BA-8327	0	0	NULL	1000	750	0.00	0.00
3	327	Down Tube	DT-2377	1	0	NULL	800	600	0.00	0.00
4	398	Handlebar Tube	HT-2981	1	0	NULL	800	600	0.00	0.00
5	399	Head Tube	HT-8019	1	0	NULL	800	600	0.00	0.00
6	847	Headlights - Dual-Beam	LT-H902	0	1	NULL	4	3	14.4334	36.3896
7	4	Headset Ball Bearings	BE-2908	0	0	NULL	800	600	0.00	0.00

Yukarıdaki fonksiyona gönderilen 'Be' parametre değeri, fonksiyon içerisinde **LIKE** ile aranacak ve bulunan sonuçlar listlenecektir.

## ÇOKLU İFADE İLE TABLO DÖNDÜREN FONKSİYONLAR

Bu fonksiyon türünün diğerlerinden farkı, içerisinde geriye değer dönmek için oluşturulan tablo tipindeki değişkene çoklu veri ekleme işlemi gerçekleştirilebilecek olunmasıdır.

En sık kullanılan sorgulardan biri, belirli aralıklardaki değerleri listelemektir. **ProductID** değeri 100 ile 500 arasında olan kayıtları listelemek buna bir örnek olabilir.

---

```
CREATE FUNCTION dbo.BelirliAraliktaUrunler(@ilk INT, @son INT)
RETURNS @values TABLE
(
    ProductID INT,
    Name      VARCHAR(30),
    ProductNumber VARCHAR(7)
)
```

```

AS
BEGIN
    INSERT @values
    SELECT ProductID, Name, ProductNumber
    FROM Production.Product
    WHERE ProductID >= @ilk AND ProductID <= @son
    RETURN
END;

```

---

**ProductID** değeri 1 ile 4 arasında olan ürünlerı listeleyelim.

---

```
SELECT * FROM dbo.BelirliAraliktaUrunler(1, 4);
```

---

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381
2	2	Bearing Ball	BA-8327
3	3	BB Ball Bearing	BE-2349
4	4	Headset Ball Bearings	BE-2908

İki parametre alan bir KTF oluşturalım. İlk parametre de, aralarında virgüler bulunan sayılar, ikinci parametrede ise bu virgüler ile ayrılan sayıları virgüllerden temizleyerek satır satır listelemek için kullanılacak bir ayıraç bulunsun.

---

```

CREATE FUNCTION dbo.IntegerAyirici(@liste      VARCHAR(8000),
                                    @ayirac     VARCHAR(10) = ',')
RETURNS @tabloDeger TABLE
(
    [Parça] INT
)
AS
BEGIN
    DECLARE @parca VARCHAR(255)
    WHILE (DATALENGTH(@liste) > 0)
        BEGIN
            IF CHARINDEX(@ayirac, @liste) > 0
                BEGIN
                    SELECT @parca = SUBSTRING(@liste,1,(CHARINDEX(@ayirac,
@liste)-1))
                    SELECT @liste = SUBSTRING(@liste,(CHARINDEX(@ayirac, @liste)

```

```

+ DATALENGTH(@ayirac)), DATALENGTH(@liste))
END
ELSE
BEGIN
    SELECT @parca = @liste
    SELECT @liste = NULL
END
INSERT @tabloDeger([Parça])
SELECT [Parça] = CONVERT(INT, @parca)
END
RETURN
END;

```

---

Fonksiyonu kullanmak için;

```
SELECT * FROM dbo.Integer_Ayirici('10, 20, 30, 300, 423, 156, 983', ',');
```

---

KTF içerisinde, tablo geri dönüş veri tipindeki nesneye birden fazla sorgu ile alınan kayıtları ekleyerek listeleyelim.

`Person.Person` tablosu içerisinde `PersonType` sütununa göre filtrelemeler gerçekleştirerek istediğimiz `PersonType` değerine sahip kayıtları listeleyelim.

Bu fonksiyon içerisinde 5 ayrı filtreden oluşan sonuç listesini birleştirerek tek bir sonuç kümesi haline getireceğiz.

	Parça
1	10
2	20
3	30
4	300
5	423
6	156
7	983

```

CREATE FUNCTION dbo.PersonTypePerson(@pt_sp VARCHAR(2), @pt_sc VARCHAR(2),
@pt_vc VARCHAR(2), @pt_in VARCHAR(2), @pt_gc VARCHAR(2))
RETURNS @PersonTypeData TABLE
(
BusinessEntityID INT,
PersonType VARCHAR(2),
FirstName VARCHAR(50),
LastName VARCHAR(50)
)
AS
BEGIN
    INSERT @PersonTypeData
    SELECT BusinessEntityID, PersonType, FirstName, LastName

```

```

FROM Person.Person
WHERE PersonType = @pt_sp
INSERT @PersonTypeData
    SELECT BusinessEntityID, PersonType, FirstName, LastName
    FROM Person.Person
    WHERE PersonType = @pt_sc
INSERT @PersonTypeData
    SELECT BusinessEntityID, PersonType, FirstName, LastName
    FROM Person.Person
    WHERE PersonType = @pt_vc
INSERT @PersonTypeData
    SELECT BusinessEntityID, PersonType, FirstName, LastName
    FROM Person.Person
    WHERE PersonType = @pt_in
INSERT @PersonTypeData
    SELECT BusinessEntityID, PersonType, FirstName, LastName
    FROM Person.Person
    WHERE PersonType = @pt_gc
RETURN
END;

```

---

Bu sorguyu daha dinamik bir şekilde geliştirerek dışarıdan gelen parametrenin daha esnek olmasını sağlayabiliyoruz. Ancak, KTF çoklu ifade ile tablo döndürme mantığını kavrayabilmek için, fazlalıklardan arındırılmış olması gerektiğini düşünüyorum.

Oluşturduğumuz fonksiyonu çağıralım.

---

```
SELECT * FROM dbo.PersonTypePerson('SP','SC','VC','IN','GC');
```

---

	BusinessEntityID	PersonType	FirstName	LastName
1	274	SP	Stephen	Jiang
2	275	SP	Michael	Blythe
3	276	SP	Linda	Mitchell
4	277	SP	Jillian	Carson
5	278	SP	Garett	Vargas
6	279	SP	Tsvi	Reiter
7	280	SP	Pamela	Anzman-Wolfe

## KULLANICI TANIMLI FONKSİYONLarda KOD GİZLİLİĞİ: ŞİFRELEMek

View ve Stored Procedure'ler de kullanılabilen kaynak kod şifreleme özelliği KTF nesneleri için de geçerlidir. Bir KTF şifrelemek de diğer nesneler gibi kolaydır.

Tüm fonksiyonların kaynak kodunun şifrelenmesine gerek yoktur. Şifrelemenin gerekli olması için, işlem yapılacak veri ve sütunların kritik öneme sahip olması ön görülür. Örneğin, ülke bilgilerinin bulunduğu bir tablo ve içeriği veriler kritik bir öneme sahip değildir. Ancak kullanıcı bilgileri, istatistik, raporlama, uygulama algoritmaları gibi önemli sayılacak veriler kritik bilgilerdir.

Daha önce geliştirdiğimiz, kullanıcı ad ve soyadını getiren fonksiyonun kaynağını şifreleyelim.

---

```
ALTER FUNCTION dbo.KullaniciGetir(@KullniciKod INT = NULL)
RETURNS VARCHAR(100)
WITH ENCRYPTION
AS
BEGIN
    DECLARE @ad_soyad VARCHAR(100)
    SELECT @ad_soyad = FirstName + ' ' + LastName
    FROM Person.Person WHERE BusinessEntityID = @KullniciKod
    RETURN @ad_soyad
END;
```

---

Şifreleme işlemi için eklediğimiz tek kod `RETURNS` komutundan sonra kullanılan `WITH ENCRYPTION` ifadesidir.

Şifrelemenin gerçekleştigini test edelim.

---

```
EXEC sp_helptext 'dbo.KullaniciGetir';
```

---

	Messages
The text for object 'dbo.KullaniciGetir' is encrypted.	

Yeni oluşturulacak (`Create`) bir fonksiyon için de aynı yöntem kullanılır.

# DETERMINİZM

Determinizm kavramı, **Deterministic** ve **Nondeterministic** olmak üzere ikiye ayrılır.

Aldığı aynı parametreler için aynı sonucu döndüren fonksiyonlar Deterministic'tir. Örneğin; 3 parametre alan ve aldığı parametreleri toplayarak geri döndüren bir fonksiyon Deterministic'tir. Çünkü aynı 3 parametreyi tekrar alduğunda aynı sonucu tekrar üretecektir. PI sayısını döndürecek bir fonksiyon da aynı şekilde Deterministic'tir.

Her çalışmasında farklı sonuç üreten fonksiyonlar Nondeterministic'tir. Sistem saatini döndüren **GETDATE()** fonksiyonu buna örnek gösterilebilir. Çünkü her çalışmasında saniye değeri aynı olsa bile, salise değeri farklı bir değer üretecektir. Bu tür fonksiyonlara örnek olarak **GUID** ve **NEWID** fonksiyonları da verilebilir. **GUID** ve **NEWID** her çalıştırıldığında farklı değerler üretirler.

Nondeterministic fonksiyonlarda küçük ama önemli bir farklılık vardır. Kimi fonksiyon her sorgu için bir kez çalışarak sonuç üretirken, kimi fonksiyon da sorgu içerisindeki her iş parçasığı için farklı değerler üretir.

**Production.Product** tablosunu inceleyelim. Bu tablodaki **rowguid** sütunu **GUID**, **ModifiedDate** sütunu ise **GETDATE** isimli Nondeterministic sistem fonksiyonları ile üretilen değerlere sahiptir.

---

```
SELECT rowguid, ModifiedDate FROM Production.Product;
```

---

	rowguid	ModifiedDate
1	694215B7-08F7-4C0D-ACB1-D734BA44C0C8	2008-03-11 10:01:36.827
2	58AE3C20-4F3A-4749-A7D4-D568806CC537	2008-03-11 10:01:36.827
3	9C21AED2-5BFA-4F18-BCB8-F11638DC2E4E	2008-03-11 10:01:36.827
4	ECFED6CB-51FF-49B5-B06C-7D8AC834DB8B	2008-03-11 10:01:36.827
5	E73E9750-603B-4131-89F5-3DD15ED5FF80	2008-03-11 10:01:36.827
6	3C9D10B7-A6B2-4774-9963-C19DCEE72FEA	2008-03-11 10:01:36.827
7	EABB9A92-FA07-4EAB-8955-F0517B4A4CA7	2008-03-11 10:01:36.827

Sorgu sonucuna bakıldığına ise, **rowguid** sütun değerlerinin tamamının benzersiz olduğunu, **ModifiedDate** sütunlarının ise aynı değerlere sahip olduğunu görebiliyoruz.

Bu durumun sebebi Nondeterministic fonksiyonlar arasındaki bu farklılıktır.

Bu tablo ve içerisindeki veriler oluşturulurken kullanılan script, tek seferde ve aynı anda çalıştırıldı. Sorgu ilk çalışırken o anın sistem tarihini alan **GETDATE** fonksiyonu tüm **INSERT** işlemleri için aynı zaman değerini kullandı. Ancak rowguid sütunu için kullanılan **GUID** fonksiyonu ise satır bazlı çalıştığından dolayı, her satır için ayrı benzersiz değer üreterek her **INSERT** işleminde yeni bir değer üretti. Bu nedenle tüm rowguid sütun değerleri farklıdır.

Ayrıca bazı Nondeterministic fonksiyonları bir skaler fonksiyon içerisinde doğrudan kullanılamaz. Örneğin, random sayısal değer üretmek için kullanılan ve her defasında farklı değer üreten **RAND** fonksiyonu bir skaler fonksiyon içerisinde doğrudan kullanılamayacaktır.

SELECT RAND();	(No column name)
1	0.457724701516271

Fonksiyon oluşturalım.

---

```
CREATE FUNCTION dbo.fnc_Rand() -- Hatalı Fonksiyon
RETURNS FLOAT
AS
BEGIN
    RETURN RAND()
END;
```

---

**fnc\_Rand** isimli fonksiyonu oluşturmak isterken hata ile karşılaştık.

Ancak bu fonksiyonu bir view içerisinde kullanarak, fonksiyon içerisinde de bu view'i çağrıdığımızda herhangi bir hata ile karşılaşmayız.

**RAND** fonksiyonunu içerisinde kullanacağımız view'i oluşturalım.

---

```
CREATE VIEW dbo.vw_Rand
AS
SELECT RAND() AS RANDOM;
```

---

Oluşturduğumuz view'i fonksiyon içerisinde kullanalım.

---

```
CREATE FUNCTION dbo.fnc_Rand()
RETURNS FLOAT
AS
BEGIN
    RETURN (SELECT * FROM dbo.vw_Rand)
END;
```

---

Fonksiyon başarılı bir şekilde oluşturuldu. Şimdi fonksiyonu çağrıabiliriz.

---

```
SELECT dbo.fnc_Rand() AS RANDOM;
```

---

	RANDOM
1	0,674615548219105

## SCHEMA BINDING

KTF nesneleri oluştururken fonksiyon içerisinde kullanılan nesnelerin, ilişkili tablolardan değiştirilmesi ya da silinmesi gibi fonksiyonun işleyişini engelleyecek işlemlerden korumak için **WITH SCHEMA BINDING** kullanılır.

**fnc\_UrunAra** isimli fonksiyonu **ALTER** ile değiştirerek fonksiyon üzerinde **SCHEMABINDING** uygulayalım.

---

```
ALTER FUNCTION fnc_UrunAra(@ara VARCHAR(10))
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT ProductID, Name FROM Production.Product
WHERE Name LIKE '%' + @ara + '%';

SCHEMABINDING ile ENCRYPTION özelliğini tek satırda belirtmek için;
WITH SCHEMABINDING, ENCRYPTION
```

---

## TABLOLARLA TABLO TİPİ FONKSİYONLARI BİRLEŞTİRMEK

Fonksiyonlar, view ve Stored Procedure'lerin bazı özelliklerini almıştır. View içerisinde **JOIN** kullanmak mümkünse de fonksiyonlar ile **JOIN** kullanımında yöntem biraz farklıdır. Yani fonksiyon ile tablonun ilişkili kullanılabilmesi, birleştirilebilmesi için **CROSS APPLY** ve **OUTER APPLY** kullanılır.

CROSS APPLY ve OUTER APPLY operatörlerini örneklendirmek için iki tablo oluşturalım.

---

Departman bilgilerini tutan Departments;

```
CREATE TABLE Departments(
    DepartmentID int NOT NULL PRIMARY KEY,
    Name VARCHAR(250) NOT NULL,
) ON [PRIMARY];
```

---

Çalışan bilgilerini tutan Employees;

```
CREATE TABLE Employees(
    EmployeesID int NOT NULL PRIMARY KEY,
    FirstName VARCHAR(250) NOT NULL,
    LastName VARCHAR(250) NOT NULL,
    DepartmentID int NOT NULL REFERENCES Departments(DepartmentID),
) ON [PRIMARY];
```

---

Oluşturduğumuz tablolara kayıt girelim.

Departmanlar;

```
INSERT Departments (DepartmentID, Name)
VALUES (1, N'Mühendislik'), (2, N'Yönetim'), (3, N'Satış'),
       (4, N'Pazarlama'), (5, N'Finans')
```

---

Çalışanlar;

```
INSERT Employees (EmployeesID, FirstName, LastName, DepartmentID)
VALUES (1, N'Kerim', N'Firat', 1), (2, N'Cihan', N'Özhan', 2),
       (3, N'Emre', N'Okumuş', 3), (4, N'Başar', N'Özhan', 3);
```

---

Eklediğimiz kayıtları listeleyerek incelelim.

```
SELECT * FROM Employees;
```

---

	EmployeesID	FirstName	LastName	DepartmentID
1	1	Kerim	Firat	1
2	2	Cihan	Özhan	2
3	3	Emre	Okumuş	3
4	4	Başar	Özhan	3

---

```
SELECT * FROM Departments;
```

---

	DepartmentID	Name
1	1	Mühendislik
2	2	Yönetim
3	3	Satis
4	4	Pazarlama
5	5	Finans

## CROSS APPLY

Tablo ve fonksiyon birleştirme işleminde, **INNER JOIN** gibi çalışan komut **CROSS APPLY** operatörüdür.

Normal bir **JOIN** ile yapılan işlemi ve **CROSS APPLY** ile nasıl yapılabileceğini inceleyelim.

### JOIN ile;

---

```
SELECT * FROM Departments D
INNER JOIN Employees E ON D.DepartmentID = E.DepartmentID;
```

---

	DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	1	Mühendislik	1	Kerim	Firat	1
2	2	Yönetim	2	Cihan	Ozhan	2
3	3	Satis	3	Emre	Okumus	3
4	3	Satis	4	Baris	Ozhan	3

### CROSS APPLY ile;

---

```
SELECT * FROM Departments D
CROSS APPLY
(
    SELECT * FROM Employees E WHERE E.DepartmentID = D.DepartmentID
) DIJIBIL;
```

---

	DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	1	Mühendislik	1	Kerim	Firat	1
2	2	Yönetim	2	Cihan	Ozhan	2
3	3	Satis	3	Emre	Okumus	3
4	3	Satis	4	Baris	Ozhan	3

İki işlemede de aynı sonucun elde edildiği görülmektedir.

## OUTER APPLY

OUTER APPLY operatörü ise LEFT OUTER JOIN gibi çalışır.

### JOIN ile;

---

```
SELECT * FROM Departments D
LEFT OUTER JOIN Employees E ON D.DepartmentID = E.DepartmentID;
```

---

	DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	1	Mühendislik	1	Kerim	Firat	1
2	2	Yönetim	2	Cihan	Özhan	2
3	3	Satis	3	Emre	Okumus	3
4	3	Satis	4	Boris	Özhan	3
5	4	Pazarlama	NULL	NULL	NULL	NULL
6	5	Finans	NULL	NULL	NULL	NULL

### OUTER APPLY ile;

---

```
SELECT * FROM Departments D
OUTER APPLY
(
    SELECT * FROM Employees E WHERE E.DepartmentID = D.DepartmentID
) KODLAB;
```

---

	DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	1	Mühendislik	1	Kerim	Firat	1
2	2	Yönetim	2	Cihan	Özhan	2
3	3	Satis	3	Emre	Okumus	3
4	3	Satis	4	Boris	Özhan	3
5	4	Pazarlama	NULL	NULL	NULL	NULL
6	5	Finans	NULL	NULL	NULL	NULL

## CROSS APPLY VE OUTER APPLY

### OPERATÖRLERİNİN FONKSİYONLAR İLE KULLANIMI

CROSS APPLY ve OUTER APPLY operatörlerinin JOIN'ler ile benzerliklerini örnekler üzerinde inceledik. Bu operatörlerin fonksiyonlar ile ilgili en önemli özelliği ise fonksiyon ile tablonun birleştirilmesi işlemidir.

Bir departmanda çalışan tüm çalışanları listelemek isteyebiliriz.

Bu işlem için bir KTF oluşturalım.

---

```
CREATE FUNCTION dbo.fnc_GetAllEmployeeOfADepartment (@DeptID AS INT)
RETURNS TABLE
AS
RETURN
(
    SELECT * FROM Employees E WHERE E.DepartmentID = @DeptID
);
```

---

Fonksiyonun **CROSS APPLY** ile kullanımı;

---

```
SELECT * FROM Departments D
CROSS APPLY dbo.fnc_GetAllEmployeeOfADepartment(D.DepartmentID);
```

---

DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	Mühendislik	1	Kerim	Firat	1
2	Yönetim	2	Cihan	Özhan	2
3	Satis	3	Emre	Okumus	3
4	Satis	4	Baris	Özhan	3

Fonksiyonun **OUTER APPLY** ile kullanımı;

---

```
SELECT * FROM Departments D
OUTER APPLY dbo.fnc_GetAllEmployeeOfADepartment(D.DepartmentID);
```

---

DepartmentID	Name	EmployeesID	FirstName	LastName	DepartmentID
1	Mühendislik	1	Kerim	Firat	1
2	Yönetim	2	Cihan	Özhan	2
3	Satis	3	Emre	Okumus	3
4	Satis	4	Baris	Özhan	3
5	Pazarlama	NULL	NULL	NULL	NULL
6	Finans	NULL	NULL	NULL	NULL

## KULLANICI TANIMLI FONKSİYONLARIN YÖNETİMİ

KTF nesnelerinin yönetimi T-SQL ya da **SSMS (SQL Server Management Studio)** ile yapılabilir. KTF nesnesinin içeriğinin değiştirilmesi, yapısal değişiklikler gibi işlemleri T-SQL ile gerçekleştirilebilir.

Aynı işlemler SSMS ile de gerçekleştirilebilir. SSMS editörü ile yapacağınız düzenleme işlemleri için gene sorgu ekranı açılacak ve oluşturma kodları, başında ALTER komutu ile birlikte listelenecaktır. Sorgu yapısının hatırlanması için iyi bir yöntemdir.

## KULLANICI TANIMLI FONKSİYONLARI DEĞİŞTİRMEK

KTF nesneleri de diğer nesnelerde olduğu gibi **ALTER** komutu ile düzenlenebilir. SSMS editörü de KTF nesnelerini düzenleyebilecek imkan sağlamaktadır.

T-SQL ile düzenlemek için;

---

```
ALTER FUNCTION dbo.BelirliAraliktakiUrunler(@ilk INT, @son INT)
RETURNS @values TABLE
(
    ProductID INT,
    Name VARCHAR(30),
    ProductNumber VARCHAR(7),
    ListPrice MONEY
)
AS
BEGIN
    INSERT @values
    SELECT ProductID, Name, ProductNumber, ListPrice
    FROM Production.Product
    WHERE ProductID >= @ilk AND ProductID <= @son
    RETURN
END;
```

---

Yukarıdaki sorgu ile **dbo.BelirliAraliktakiUrunler()** fonksiyonun düzenleme işlemini gerçekleştirdik.

Bu düzenleme işleminde yapılan iki ana işlem var.

- **CREATE FUNCTION** yerine **ALTER FUNCTION** kullanıldı.
- **SELECT** sorgu yapısında değişiklik yapıldı. Bu nedenle, **RETURNS** ile döndürülecek tablo yapısı da düzenlenendi.

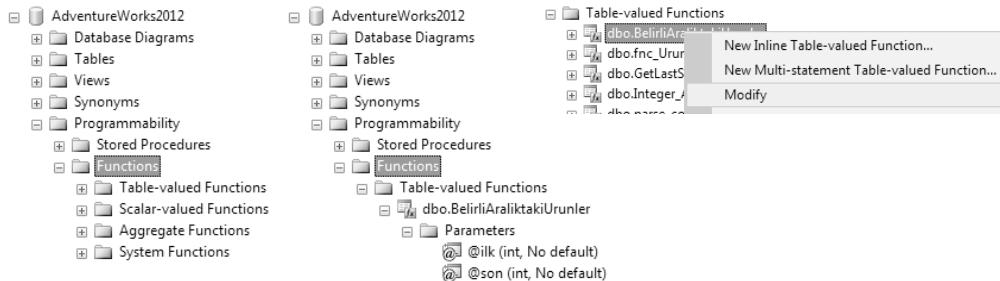
**SELECT** sorgusuna **ListPrice** sütununu eklediğimiz için **RETURNS** kısmında da bu sütunu aşağıdaki şekilde tanımladık.

---

```
...
ListPrice MONEY
...
```

---

SSMS ile KTF Düzenleme;



SSMS ya da T-SQL ile KTF düzenleyebilmek için, KTF'nin **WITH ENCRYPTION** ile şifrelenmemiş olması gereklidir. Şifrelenmiş bir KTF nenesinin içeriğine ulaşılamayacağı için düzenleme işlemi gerçekleştirilemez.

## KULLANICI TANIMLI FONKSİYONLARI SİLMEK

Fonksiyon silmek için **DROP** komutu kullanılır.

---

```
DROP FUNCTION [dbo].[BelirliAraliktaUrunler]
```

---

T-SQL ile silinebileceği gibi SSMS ile de görsel olarak silinebilir.

SSMS ile silebilmek için;

**Programmability > Functions** içerisinde ilgili kısımdaki fonksiyona sağ tıklayarak, **Delete** seçeneği ile silebilirsiniz.



# SQL SERVER İLE XML

15

Teknolojide birçok standart ve yeni platformların gelişmesiyle farklı veri formatları ve platformlar arasında veri uyumluluğu, veri transferi gibi kavramlar önem kazandı. İki farklı nesne yapısının birbiri ile iletişime geçmesi için ara katman oluşturmak gereklidir. Veriyi farklı bir formata dönüştürecek bir parser bu işi görebilir. Ancak veri dönüşümünün birden fazla formatlara yapılması gereken durumlarda söz konusu olabilir.

Bir e-ticaret sistemini düşünelim. Bu tür sistemler, hizmet sağladığı firmalara bazı verileri iletmeli. Anlaşmalı olduğu kargo firmasına sipariş edilen ve kargoya hazır hale gelen ürünleri iletmek için ilgili veriyi belirli bir formata dönüştürerek kargo firmasına ullaştırır. Kargo firması da bu veriyi kendi yazılım mimarisine göre alarak işler ve veritabanlarına kaydeder. Aynı şekilde e-ticaret sistemi de tüm yayın evlerinin yayındaki kitaplarının ya da beyaz eşya firmalarındaki ürünlerin satış fiyatı, ürün adı, ürün açıklaması, ürün resmi gibi tüm bilgileri belirli formatta alarak e-ticaret veritabanına kaydeder. Fark ettiyseniz, benzer işlemler için birbirinden farklı platformların entegrasyonu söz konusu oldu. Bu sistemde entegrasyon yapan firmaların sayısı onlarca olabilir. Ve entegrasyona katılan tüm firmaların yazılımları, veritabanları, işletim sistemleri, kullandıkları teknolojiler tamamen farklı olabilir. Aynı şekilde bu firmaların tamamı farklı veri formatları ile çalışıyor olabilir. Bazıları XML kullanıyor olabileceği gibi farklı formatlarda da veri aktarımı yapıyor olabilir.

XML, bu tür farklı platform ve teknolojilerin bir arada uyumlu olarak veri aktarımı, transferi yapabilmesi için geliştirilen bir standarttır.

XML'i İngilizce diline benzetebiliriz. Siz hangi ülke ve milletten olursanız olun diliniz farklı da olsa, Dünya'da en yaygın dil olarak kullanılan İngilizce'yi bilen tüm insanlarla konuşabilirsınız. İngilizce lisans dili alanında bir standart ise, XML'de bilgisayar biliminde veri aktarımı, uyumluluk gibi konularda farklı platformları tek bir standart ile iletişim halinde tutmayı sağlayan bir teknolojidir.

Büyük veritabanı yönetim sistemlerinin tamamında desteklendiği gibi SQL Server'da XML veri formatına varsayılan olarak destek vermektedir ve tam uyumlu şekilde çalışmaktadır.

## XML

**XML** (*eXtensible Markup Language*) standarı **W3C** (*World Wide Web Consortium*) tarafından standart haline getirilen ve HTML'in de tasarımcısı olan Tim Berners Lee tarafından tasarlanmış bir işaretleme dilidir. XML'in Türkçe anlamı Genişletilebilir İşaretleme Dili'dir.

XML'de veriler tam anlamıyla genişletilebilir şekilde etiketleme sistemi ile işaretlenir.

Örneğin, KodLab yayinevine ait kitaplar XML formatında şu şekilde tutulabilir.

---

```
<root>
  <kitap kitapID=123> İleri Seviye SQL Server T-SQL </kitap>
  <kitap kitapID=124> İleri Seviye Android Programlama </kitap>
<root>
```

---

XML büyük-küçük harf duyarlı bir işaretleme dilidir. **<KodLab>** ile **<kodlab>** aynı değildir. Her XML dokümanında bir kök dizin olmak zorundadır. Kök dizin içerisinde ise kök dizine ait elemanlar bulunur. **<kitap>** bu örnekte bir alt elemandır. **<kitap>** içerisindeki **kitapID** ise bir **attribute** (öznitelik)'tür.

## XML VERİ TIPIΝI KULLANMAK

XML'in bir standart haline gelmesinden sonra SQL standartları tarafından desteklenir hale gelmesiyle SQL Server'da gelişmiş seviyede XML desteği sunar. XML metodları, veri tipi ve şema gibi birçok özelliğiyle birlikte desteklenen XML ile SQL Server'da şu şekilde faydalanaılabilir.

- XML tipinde bir değişken tanımlamak için kullanılabilir.
- Tablo oluştururken XML tipinde sütunlar oluşturulabilir.
- Stored Procedure'lere girdi-çıktı parametreleri olarak kullanılabilir.
- Kullanıcı tanımlı fonksiyonlara girdi parametre ya da geri dönüş tipi olarak kullanılabilir.

XML, yapısal olarak diğer veri tiplerinden farklıdır. Bazı durumlarda farklı veri tiplerine dönüştürülmesi gereklidir. Dönüşüm işlemi için `CAST` ya da `CONVERT` fonksiyonları kullanılabilir.

XML, **tip tanımlı** (*typed*) ve **tip tanımsız** (*untyped*) olarak ikiye ayrılır. Bu bölümde bu iki XML tipi için de detaylı örnekler yaparak, normal sorgu ve Stored Procedure ile kullanımlarını inceleyeceğiz.

## XML TİPİ İLE DEĞİŞKEN VE PARAMETRE KULLANMAK

SQL Server, XML teknolojisine native olarak destek vermektedir. XML bir veri tipi kullanılabileceği gibi, bir değişken olarak da kullanılabilir. XML'in değişken ve parametre olarak kullanılabilmesi, SQL Server veritabanı programlama esnekliğini ve gücünü tam olarak kullanabilmek anlamına gelir.

Örnek ve uygulamalarda birçok farklı şekilde kullanılacak XML'in değişken olarak nasıl kullanılabileceğine basit bir örnek verelim.

---

```
DECLARE @xml_veri VARCHAR(MAX);
SET @xml_veri = '
<kitaplar>
    <kitap>İleri Seviye SQL Server T-SQL</kitap>
    <kitap>İleri Seviye Android Programlama</kitap>
    <kitap>Java SE</kitap>
</kitaplar>';
SELECT CAST(@xml_veri AS XML);
SELECT CONVERT(XML, @xml_veri);
PRINT @xml_veri;
```

---

(No column name)
1 <kitaplar><kitap>İleri Seviye SQL Server T-SQL</kitap><kitap>İleri Seviye Android Programlama</kitap><kitap>Java SE</kitap></kitaplar>
(No column name)
1 <kitaplar><kitap>İleri Seviye SQL Server T-SQL</kitap><kitap>İleri Seviye Android Programlama</kitap><kitap>Java SE</kitap></kitaplar>

```
(1 row(s) affected)

(1 row(s) affected)

<kitaplar>
  <kitap>İleri Seviye SQL Server T-SQL</kitap>
  <kitap>İleri Seviye Android Programlama</kitap>
  <kitap>Java SE</kitap>
</kitaplar>
```

Metin olarak oluşturulan XML veri değişkeni, sonrasında **CAST** ve **CONVERT** ile XML veri tipine dönüştürülmektedir.

**VARCHAR** olarak tanımlanan verinin XML veri tipine dönüştürülmesi bir seçenektedir. Değişken, doğrudan XML olarak da aşağıdaki gibi tanımlanabilir.

---

```
DECLARE @xml_veri XML
SET @xml_veri = '
<kitaplar>
  <kitap>İleri Seviye SQL Server T-SQL</kitap>
  <kitap>İleri Seviye Android Programlama</kitap>
  <kitap>Java SE</kitap>
</kitaplar>'

SELECT CAST(@xml_veri AS VARCHAR(MAX))
SELECT CONVERT(VARCHAR(MAX), @xml_veri)
PRINT CONVERT(VARCHAR(MAX), @xml_veri)
```

---

	(No column name)
1	<kitaplar><kitap>İleri Seviye SQL Server T-SQL</kitap><kitap>İleri Seviye Android Programlama</kitap><kitap>Java SE</kitap></kitaplar>
	(No column name)
1	<kitaplar><kitap>İleri Seviye SQL Server T-SQL</kitap><kitap>İleri Seviye Android Programlama</kitap><kitap>Java SE</kitap></kitaplar>

```
(1 row(s) affected)

(1 row(s) affected)
<kitaplar><kitap>İleri Seviye SQL Server T-SQL</kitap><kitap>İleri Seviye Android Programlama</kitap><kitap>Java SE</kitap></kitaplar>
```

## TİP TANIMSIZ XML VERİ İLE ÇALIŞMAK (UNTYPED)

XML verinin şeması ya da yapısının denetlenmemesi gereken durumlarda kullanılır. Tip tanimsız kullanım ile XML veri tipine sahip sütun, sadece verinin XML olup olmadığını denetlemek için kullanılır. XSD tanımlaması, XML yapısının SQL Server tarafından tanınmasını sağlar. SQL Server, XML yapıyı ne kadar iyi tanırsa sorgu performansını buna göre optimize ederek daha performanslı XML kullanımı gerçekleştirilmesini sağlar. Tip tanimsız XML kullanımında bu tanımlamaların bulunmaması performansı olumsuz yönde etkileyecektir.

Tip tanımsız XML veri içeren bir tablo oluşturalım.

---

```
CREATE TABLE OzGecmis
(
    AdayID INT IDENTITY PRIMARY KEY,
    AdayOzGecmis XML
);
```

---

**HumanResources.JobCandidate** içerisinde, iş başvurusu yapan adayların kayıtları bulunur. Adayların özgeçmiş bilgilerinin yapısından dolayı, özel bir XML olarak tutmak daha kolay bir kullanımdır. Aday öz geçmiş bilgileri çok önemli ve sık kullanılacak veriler olmadığı için yüksek sorgu performansına da ihtiyacı yoktur. Bu nedenle tip tanımlı ya da tanımsız olması sorun teşkil etmeyecektir.

**OzGecmis** tablosundaki **AdayOzGecmis** sütununa, **HumanResources.JobCandidate** tablosundaki **Resume** sütunundan tüm kayıtları seçerek ekleyelim.

---

```
INSERT INTO OzGecmis (AdayOzGecmis)
SELECT Resume FROM HumanResources.JobCandidate;
```

---

**OzGecmis** tablosunun içerik eklendikten sonraki halini görüntüleyelim.

---

```
SELECT * FROM OzGecmis;
```

---

	AdayID	AdayOzGecmis
1	1	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>M.</ns:Name Prefix><...>
2	2	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>M.</ns:Name Prefix><...>
3	3	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>M.</ns:Name Prefix><...>
4	4	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Peng...>
5	5	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Shen...>
6	6	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Tai</...>
7	7	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>Thierry</ns:Name Prefix><...>

**AdayID** değeri 1 olan adayın özgeçmişini görüntüleyelim.

---

```
DECLARE @Aday XML;
SELECT @Aday = AdayOzGecmis FROM OzGecmis WHERE AdayID = 1;
SELECT @Aday AS Resume;
```

---

Resume
<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>M.</ns:Name Prefix><ns:Name First>Thierry</ns:Name First><ns:Name Middle>Thierry</ns:Name Middle><ns:Name Surname>Thierry</ns:Name Surname></ns:Name></ns:Resume>

Görüntülenen aday XML dokümanındaki Name alanı aşağıdaki gibidir.

---

```
<ns:Name>
  <ns:Name.Prefix>M.</ns:Name.Prefix>
  <ns:Name.First>Thierry</ns:Name.First>
  <ns:Name.Middle />
  <ns:Name.Last>D'Hers</ns:Name.Last>
  <ns:Name.Suffix />
</ns:Name>
```

---

XML veri tipine sahip bir tablo oluşturup, XML sonuç dönen sorgular ile içerikleri görüntüleyebildik. Genel olarak bu tür işlemler prosedürel olarak gerçekleştirilebilir. Prosedür içerisinde gerekli düzenleme ve filtrelemeler yapılabildiği ve her seferinde uzun kod satırları yazmaya gerek olmadığı için prosedürler XML gibi büyük ve karmaşık veriler için kullanılabilir nesnelerdir.

Yukarıda yapılan sorguyu Stored Procedure ile gerçekleştirelim.

---

```
CREATE PROCEDURE AdayEkle( @Aday XML )
AS
  INSERT INTO OzGecmis(AdayOzGecmis) VALUES(@Aday);
```

---

**AdayEkle** prosedürü, dışarıdan adayların XML bilgilerini alarak **OzGecmis** tablosuna kaydediyor. Şimdi, bir adayın XML bilgilerini **AdayEkle** prosedürü ile **OzGecmis** tablosuna kayıt edelim.

---

```
DECLARE @AdayProc XML;
SELECT @AdayProc = Resume FROM HumanResources.JobCandidate
  WHERE JobCandidateID = 8;
EXEC AdayEkle @AdayProc;
```

---

**JobCandidateID** değeri 8 olan adayın özgeçmiş bilgilerini **AdayEkle** prosedürünü kullanarak **OzGecmis** tablosuna kayıt ettik.

Son kayıt eklendikten sonra **OzGecmis** tablosunun görünümü aşağıdaki gibi olacaktır.

---

```
SELECT * FROM OzGecmis ORDER BY AdayID DESC;
```

---

	AdayID	AdayOzGecmis
1	10	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Peng</ns:Name ...
2	9	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Ynu </ns:Name ...
3	8	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>yanu </ns:Name Prefix><ns:Name ...
4	7	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>yanu </ns:Name Prefix><ns:Name ...
5	6	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix>tai </ns:Name Prefix><ns:Name ...
6	5	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Shengda</ns:Na ...
7	4	<ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix /><ns:Name First>Peng</ns:Name ...

## TİP TANIMLI XML VERİ İLE ÇALIŞMAK (TYPED)

Tip tanımlı XML kullanımında, bir XML veri, XML şema tanımları (XSD) ile denetlenir. XML şemaları, verinin yapısını belirtir ve kısıtlamalar, hangi sütuna ne türden veri girileceği gibi denetleme şartlarını sağlayan performans artırıcı özelliklerdir. XML verisini SQL Server'a doğru tanıtabacağı için performans olarak etkili bir kullanımdır.

Tip tanımlı XML için veritabanında XML şema koleksiyonu oluşturulması gereklidir. XML şema koleksiyonlarını detaylarıyla anlattığımız kısmı inceleyebilirsiniz.

Tip Tanımsız XML konusunu işlerken verdiğimiz **OzGecmis** tablosu örneğini, Tip Tanımlı XML yöntemi ile nasıl oluşturacağımızı ve yöneteceğimizi inceleyelim. Tabloyu yeniden oluşturmak için **DROP TABLE** ile silin.

---

```
CREATE TABLE OzGecmis
(
    AdayID INT IDENTITY PRIMARY KEY,
    AdayOzGecmis XML (HumanResources.HRResumeSchemaCollection)
);
```

---

Tip tanımlı XML kullanımında önemli bir farklılık, **AdayOzGecmis** sütunundaki XML veri tipinin yanında, parantez içerisinde bir de XML şema koleksiyonu kullanılmasıdır.

Kullanılan XML şema koleksiyonu **HumanResources** şeması içerisindeki **HRResumeSchemaCollection**'dur.

XML şema koleksiyonlarını işlediğimiz bölümü detaylı inceleyerek bu konuyu daha iyi kavrayabiliyoruz.

**OzGecmis** tablosuna **SELECT** ile veri girişini gerçekleştirelim.

---

```
INSERT INTO OzGecmis (AdayOzGecmis)
SELECT Resume FROM HumanResources.JobCandidate;
```

---

Eklenen kayıtları görmek için tabloyu görüntüleyelim.

```
SELECT * FROM OzGecmis;
```

Değişken tanımlayarak **AdayID** değeri 8 olan kaydı görüntüleyelim.

```
DECLARE @Aday XML (HumanResources.HRResumeSchemaCollection);
SELECT @Aday = AdayOzGecmis FROM OzGecmis WHERE AdayID = 8;
SELECT @Aday AS Aday;
```

```
1 <ns:Resume xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"><ns:Name><ns:Name Prefix="muu" /><ns:Name Prefix="ns" Name="First">muu</ns:
```

Dikkat edilmesi gereken bir konu, XML şema koleksiyonun sadece tablo oluştururken veri tipinin yanında kullanılmamış olması. Aynı şekilde **DECLARE** ile tanımlanan değişkenin veri tipinin yanında da XML şema koleksiyonunu belirtmemiz gerekti.

Tip tanımlı XML örneğini prosedür kullanarak otomatik hale getirelim.

```
ALTER PROCEDURE AdayEkle
    @Aday XML (HumanResources.HRResumeSchemaCollection)
AS
    INSERT INTO OzGecmis (AdayOzGecmis)
    VALUES (@Aday);
```

**AdayEkle** prosedürümüzü kullanalım.

```
DECLARE @AdayProc XML;
SELECT @AdayProc = Resume FROM HumanResources.JobCandidate WHERE
JobCandidateID = 8;
EXEC AdayEkle @AdayProc;
```

**JobCandidateID** değeri 8 olan adayın XML özgeçmişini prosedür kullanarak **OzGecmis** tablosuna ekledik.

---

```
SELECT * FROM OzGecmis;
```

---

## XML VERİ TİPİ İLE ÇOKLU VERİ İŞLEMLERİ

Veri tipi XML olan bir sütun tanımlamak birçok durumda gereklidir. Bazen, SQL Server tablo yapısında genel ve tüm ürünlerde ortak olan ürünlerin bilgilerini tutmak, geri kalan birbirinden farklı ürün özelliklerini de XML şema koleksiyonları denetiminde, veritabanında XML olarak tutmak istenebilir.

XML veri tipi ve diğer SQL Server veri tiplerinden oluşan bir tablo oluşturalım. Bu tabloda, kitap bilgileri yer alınsın. Kitapların sadece **KitapID** ve kitap isimlerini tutan iki adet XML olmayan sütun bulunsun. Ek olarak, birden fazla sütun değeri içerecek XML veri tipine sahip sütun bulunsun.

---

```
CREATE TABLE Kitaplar
(
    KitapID INT IDENTITY(1,1) PRIMARY KEY,
    Ad VARCHAR(60),
    KitapDetay XML
);
```

---

Kitaplar tablosunda temel bazı özellikler haricinde sadece **KitapDetay** sütunu var. **KitapDetay** sütunu, Kitaplar tablosuna esneklik katmaktadır. Yapısal olarak birçok farklı veri tek bir sütuna XML olarak eklenebilir.

**KitapDetay** sütunu, XML şema koleksiyonu oluşturularak bu koleksiyonlar ile denetlenebilir.

Çoklu veri ekleme yöntemi ile Kitaplar tablosuna iki kitap kaydı ekleyelim.

---

```
INSERT INTO Kitaplar(Ad, KitapDetay)
VALUES('İleri Seviye SQL Server T-SQL',
'<Kitap>
<Yazar>Cihan Özhan</Yazar>
<ISBN>978-975-17-2268-7</ISBN>
<Ozet>İleri seviye SQL Server kitabı.</Ozet>
<SayfaSayisi>500</SayfaSayisi>')
```

```

<BaskiSayisi>3</BaskiSayisi>
</Kitap>'),
('İleri Seviye Android Programlama',
'<Kitap>
<Yazar>Kerim FIRAT</Yazar>
<ISBN>978-975-17-2243-8</ISBN>
<Ozet>İleri seviye Android programlama kitabı.</Ozet>
<SayfaSayisi>800</SayfaSayisi>
<BaskiSayisi>5</BaskiSayisi>
</Kitap>');

```

---

Eklenen veriyi görüntüleyelim.

---

```
SELECT * FROM Kitaplar;
```

---

	KitapID	Ad	KitapDetay
1	1	İleri Seviye SQL Server T-SQL	<Kitap><Yazar>Cihan Özhan</Yazar><ISBN>978-975-17-2268-7</ISBN><Ozet>İleri seviye SQL Server kitabı.</Ozet><SayfaSayisi>...</SayfaSayisi>
2	2	İleri Seviye Android Programlama	<Kitap><Yazar>Kerim FIRAT</Yazar><ISBN>978-975-17-2243-8</ISBN><Ozet>İleri seviye Android programlama kitabı.</Ozet><Say...</SayfaSayisi>

Kitap ekleme işlemini prosedürel hale getirelim.

---

```

ALTER PROC KitapEkle
(
    @Ad  VARCHAR(60),
    @KitapDetay XML
)
AS
BEGIN
    INSERT INTO Kitaplar(Ad, KitapDetay)
    VALUES(@Ad, @KitapDetay)
END;

```

---

KitapEkle isimli prosedürü kullanarak veritabanına bir kayıt ekleyelim.

---

```

DECLARE @KD XML;
SET @KD = '<Kitap>
<Yazar>Kerim FIRAT</Yazar>
<ISBN>978-975-17-2243-8</ISBN>
<Ozet>Java Standard Edition eğitim kitabı.</Ozet>
<SayfaSayisi>700</SayfaSayisi>
<BaskiSayisi>4</BaskiSayisi>
</Kitap>';
EXEC KitapEkle 'Java SE', @KD;

```

---

# XML ŞEMA KOLEKSİYONLARI

XML standardının ilk geliştiği yıllarda **DTD** (*Document Type Definition*) kullanılıyordu. DTD yapısal olarak zor ve karmaşık bir kullanıma sahipti.

## DTD

**DTD** (*Document Type Definition*), XML dokümanlarının yapısal kurallarını belirleyen bir yapı sağlar. Bir XML içerisinde hangi elementlerin ve her elementin içerisinde kaç attribute'ü olacağını belirmek için kullanılır.

## DOCTYPE BİLDİRİMİ

XML dokümanlarının hangi DTD'ye uyacağını belirtmek için kullanılır. XML'de bildirim işlemi <!> ve < > tag açma-kapama işaretleri ile gerçekleştirilir.

Bir DTD dokümanının uzantısı dtd olmalıdır.

Aşağıda, dijibil root elementine sahip bir XML dokümanı için dijibil.dtd bildirimini yapılmaktadır.

---

```
<!DOCTYPE dijibil SYSTEM "http://www.dijibil.com/dtds/dijibil.dtd">
```

---

Bu işlemde ilk olarak root elementin adı verilir. Eğer daha önce yapılmış bir DTD kullanılmak isteniyorsa, **SYSTEM** yerine **PUBLIC** kullanılır.

DTD dört tip bildirim içerir.

- ELEMENT
- ATTLIST
- ENTITY
- NOTATION

## ELEMENT BİLDİRİMİ

XML dokümanlarında kullanılacak elementleri tanımlamak için kullanılır.

---

```
<!ELEMENT isim CATALOG>
```

---

ya da

---

```
<!ELEMENT isim (content)>
```

---

İçinde hiç bir şey kullanılmayan bir element için;

---

```
<!ELEMENT br EMPTY>
```

---

Bir element herhangi bir element içerebilirse **EMPTY** yerine **ANY** kullanılır.

---

```
<!ELEMENT br ANY>
```

---

XML içerisinde açıklama satırı gibi yazmış iseniz **PCDATA** komutuna aşina olmalısınız.

Bir elementin içinde sadece yazı olması gerekiyorsa;

---

```
<!ELEMENT soyisim (#PCDATA)>
```

---

XML içerisinde birden fazla element kullanmak için;

---

```
<!ELEMENT dijibil (isim, soyisim)>
```

---

Çoklu element kullanımında SQL Server sütunlarında olduğu gibi sıralama önemlidir.

XML içerisinde bir element kullanılırsa diğerinin kullanılmaması istendiğinde;

---

```
<!ELEMENT dijibil (isim|soyisim)>
```

---

## ATTRLIST BİLDİRİMİ

Elementlerin attribute tanımlamaları için kullanılır.

**Söz Dizimi:**

---

```
<!ATTLIST element_ismi
      attribute_ismi attribute_tipi
      attribute_varsayılan varsayılan_değer>
```

---

İki attribute'ü olan dijibil root elementini tanımlayalım.

---

```
<!ATTLIST dijibil
      UserID CDATA #REQUIRED
      UserNO  CDATA #IMPLIED>
```

---

- **CDATA**: Karakter verisi anlamına gelir.
- **#REQUIRED**: Bu attribute'ün kullanımının zorunlu olduğunu belirtir.
- **#IMPLIED**: Bu attribute'ün kullanımının zorunlu değil, istege bağlı olduğunu belirtir.

Attribute'ler istenen değerleri seçenek olarak sunabilir. Bu şekilde kullanılarak attribute'ün sadece belirli değerleri alması sağlanabilir.

Genel olarak yazılımlarda cinsiyet bilgileri tutulur. Bu bilgileri, XML ortamındaki attribute'lerde aşağıdaki gibi seçenekli hale getirilebilir.

---

```
<!ATTLIST kullanici cinsiyet (bay | bayan) #IMPLIED>
```

---

Yukarıdaki kullanım ile kullanıcının, bu attribute'e sadece 'bay' ya da 'bayan' değerleri vermesi sağlanmış olur.

Attribute'lerde varsayılan değer ataması ise şu şekilde gerçekleşir.

---

```
<!ATTLIST kullanici cinsiyet (bay | bayan) "bay" #IMPLIED>
```

---

Yukarıdaki varsayılan değer örneğinde, iki seçimiği değer için varsayılan olarak 'bay' değeri atanacaktır.

## **XML ŞEMA KOLEKSİYONLARI HAKKINDA BİLGİ ALMAK**

XML şemaları hakkında bilgi almak için `xml_schema_collections` sistem kataloğunu kullanabilirsiniz. Bu katalog ile üstünde çalışığınız veritabanında var olan şema koleksiyonlarını listeleyebilirsiniz.

Kullandığımız `AdventureWorks` veritabanına ait şema kataloglarını listeleyelim.

---

```
SELECT * FROM sys.xml_schema_collections;
```

---

	xml_collection_id	schema_id	principal_id	name	create_date	modify_date
1	1	4	NULL	sys	2009-04-13 12:59:13.390	2011-11-04 21:07:51.970
2	65536	6	NULL	AdditionalContactInfoSchemaCollection	2012-03-14 13:14:18.920	2012-03-14 13:14:18.930
3	65537	6	NULL	IndividualSurveySchemaCollection	2012-03-14 13:14:18.953	2012-03-14 13:14:18.953
4	65538	5	NULL	HRResumeSchemaCollection	2012-03-14 13:14:18.980	2012-03-14 13:14:18.980
5	65539	7	NULL	ProductDescriptionSchemaCollection	2012-03-14 13:14:19.027	2012-03-14 13:14:19.037
6	65540	7	NULL	ManuInstructionsSchemaCollection	2012-03-14 13:14:19.060	2012-03-14 13:14:19.060
7	65541	9	NULL	StoreSurveySchemaCollection	2012-03-14 13:14:19.110	2012-03-14 13:14:19.110

XML yapısında önemli bir yere sahip isim uzayları (*namespace*) hakkında bilgi almak için `xml_schema_namespaces` sistem kataloğunu kullanabilirsiniz.

---

```
SELECT * FROM sys.xml_schema_namespaces;
```

---

	xml_collection_id	name	xml_namespace_id
1	1	http://www.w3.org/2001/XMLSchema	1
2	1	http://schemas.microsoft.com/sqlserver/2004/sqltyp...	2
3	1	http://www.w3.org/XML/1998/namespace	3
4	65536	http://schemas.microsoft.com/sqlserver/2004/07/a...	1
5	65536	http://schemas.microsoft.com/sqlserver/2004/07/a...	2
6	65536	http://schemas.microsoft.com/sqlserver/2004/07/a...	3
7	65537	http://schemas.microsoft.com/sqlserver/2004/07/a...	1

## XML\_SCHEMA\_NAMESPACE İLE ŞEMA KOLEKSİYONLARINI LISTELEMEK

XML şema koleksiyonları karmaşık bir yapıya sahiptir. SQL Server yeteneklerini kullanarak bir XML şema koleksiyona erişebilmek önemli bir gereksinimdir.

Şimdi, bazı örnekler yaparak `AdventureWorks` veritabanı ile ilgili bazı şemaların XML içeriklerini listeleyelim.

---

`Production.ProductDescriptionSchemaCollection` şema koleksiyonunu görüntüleyelim.

---

```
SELECT
xml_schema_namespace(
N'Production',
N'ProductDescriptionSchemaCollection');
```

---

Sonuç ekranındaki XML bağlantısı açıldığında, şema koleksiyonunun XML içeriğini görüntülenecektir.

XQuery konusuna daha sonra değineceğiz. Ancak temel olarak kullanımı görmemiz için bir şema koleksiyonu XQuery ile listeleyelim.

### Söz Dizimi:

---

```
SELECT xml_schema_namespace (
    N'SchemaName',
    N'XmlSchemaCollectionName')
    .query ('/xs:schema[@targetNamespace="TargetNameSpace"]');
```

---

Production şeması içerisindeki `ProductDescriptionSchemaCollection` isimli şema koleksiyonunu görüntüleyelim.

---

```
SELECT xml_schema_namespace (
    N'Production',
    N'ProductDescriptionSchemaCollection').query('
/xs:schema[@targetNamespace="http://schemas.microsoft.com/
sqlserver/2004/07/adventure-works/ProductModelWarrAndMain"]');
```

---

Aynı işlemi farklı bir şekilde de gerçekleştirebiliriz.

---

```
SELECT xml_schema_namespace (
    N'Production',
    N'ProductDescriptionSchemaCollection', N'http://schemas.microsoft.
com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain');
```

---

Var olan XML şema koleksiyonları görüntülemeyi öğrenmek, kendi XML şema koleksiyonlarını oluşturmayı ve yönetmeyi daha kolay hale getirdi. Artık kendi koleksiyonlarımızı oluşturarak yönetebiliriz.

## XML ŞEMA KOLEKSİYONU OLUŞTURMAK

Tüm nesne oluşturma söz dizimleri bu işlem için de geçerlidir. `CREATE` ile bir şema koleksiyonu oluşturulabilir.

### Söz Dizimi:

---

```
CREATE XML SCHEMA COLLECTION [sql_server_sema,] koleksiyon_ismi
AS { sema_metni | sema_metnini_iceren_degisken }
```

---

## Örnek bir Söz Dizimi:

---

```
CREATE XML SCHEMA COLLECTION EmployeeSchema
AS'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element >
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element />
      <xsd:element />
      <xsd:element />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>'
```

---

XML şema koleksiyonu oluşturmak için **AdventureWorks** veritabanında **Production** şeması içerisindeki **ProductDescriptionSchemaCollection** isimli şema koleksiyonunun XML kodunu elde ederek farklı bir isimde tekrar oluşturabiliriz.

XML kaynağı elde etmek için;

---

```
SELECT xml_schema_namespace(
N'Production',
N'ProductDescriptionSchemaCollection', N'http://schemas.microsoft.
com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain');
```

---

Bir bağlantı şeklinde gelen sonuca tıklayarak görüntülenen XML kodu kopyalıyoruz. Kopyalanan kodu aşağıdaki şema koleksiyonu oluşturma kodunun sonunda bulunan iki tek tırnak arasına yapıştırıyoruz.

---

```
CREATE XML SCHEMA COLLECTION yeniXMLSemaKoleksiyon AS ''
```

---

Kod görünümü aşağıdaki gibi olacaktır. Kodlar seçili hale getirilip çalıştırıldığında **yeniXMLSemaKoleksiyon** isminde yeni bir XML şema koleksiyonuna sahip oluruz.

---

```
CREATE XML SCHEMA COLLECTION yeniXMLSemaKoleksiyon AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/
```

```

adventure-works/ProductModelWarrAndMain" targetNamespace="http://
schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelWarrAndMain" elementFormDefault="qualified">
<xsd:element name="Maintenance">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:sequence>
          <xsd:element name="NoOfYears" type="xsd:string" />
          <xsd:element name="Description" type="xsd:string" />
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Warranty">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:sequence>
          <xsd:element name="WarrantyPeriod" type="xsd:string" />
          <xsd:element name="Description" type="xsd:string" />
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
</xsd:schema>';

```

---

**Yeni şema koleksiyonumuzu sorgulamak için;**

---

```

SELECT xml_schema_namespace(
  N'dbo',N'yenixMLSemaKoleksiyon', N'http://schemas.microsoft.com/
sqlserver/2004/07/adventure-works/ProductModelWarrAndMain');

```

---

**Şema koleksiyonumuz varsayılan olarak dbo şeması içerisinde oluşturulduğu için **SELECT** sorgusunda **dbo** şema adını belirttik.**

## XML ŞEMA KOLEKSİYONU DEĞİŞTİRMEK

XML şema koleksiyonunu değiştirmek komut olarak diğerleriyle aynıdır. **ALTER** ifadesi ile değişiklik gerçekleştirilebilir. Ancak diğer nesnelere göre şema koleksiyonlarında **ALTER** işlemi bir kısıtlamaya sahiptir. Şema koleksiyonlarında **ALTER** sadece yeni kısımlar eklemek için kullanılabilir.

### Söz Dizimi:

---

```
ALTER XML SCHEMA COLLECTION [sql_server_sema,] koleksiyon_ismi
ADD { sema_metni | sema_metnini_iceren_degisken }
```

---

## XML ŞEMA KOLEKSİYONUNU KALDIRMAK

XML şema koleksiyonlarını kaldırmak diğer nesneler gibi gerçekleştirilir.

### Söz Dizimi:

---

```
DROP XML SCHEMA COLLECTION [ sql_server_sema .] koleksiyon_ismi
```

---

Oluşturduğumuz **yeniXMLSemaKoleksiyon** isimli şema koleksiyonunu kaldırıralım.

---

```
DROP XML SCHEMA COLLECTION yeniXMLSemaKoleksiyon;
```

---

## XML VERİ TIPI METODLARI

XML veri üzerinde işlemler gerçekleştirilecek bazı metodlara sahiptir. Bu metodlar benzersiz özelliklere sahip olduğu gibi, kendi içlerinde söz dizimleri de farklılık gösterir. XML verinin sorgulanması, işlenmesi ve dönüştürülmesi için fonksiyonel özellikler sağlar.

XML metodları için kullanacağımız tablo ve kayıtları oluşturalım.

---

```
CREATE TABLE Magazalar
(
    MagazaID   INT PRIMARY KEY,
    Anket_UnTyped XML,
    Anket_Typed XML(Sales.StoreSurveySchemaCollection)
);
```

---

**Magazalar** ismindeki tablonun XML veri tipindeki **Anket\_Typed** sütununda şema koleksiyonu olarak **Sales.StoreSurveySchemaCollection**'u kullanıyoruz.

**Magazalar** tablosuna, **AdventureWorks** veritabanındaki **Sales.Store** tablosundan veri aktaracağız.

---

```
INSERT INTO Magazalar
VALUES
(292,'<MagazaAnket>
<YillikSatis>145879</YillikSatis>
<YillikGelir>79277</YillikGelir>
<BankaAd>HIC Bank</BankaAd>
<IsTuru>CO</IsTuru>
<AcilisYil>2005</AcilisYil>
<Uzmanlik>Technology</Uzmanlik>
<Markalar>2</Markalar>
<Internet>ISDN</Internet>
<CalisanSayisi>14</CalisanSayisi>
<Urunler Tip="Yazilim">
    <Urun>Mobil</Urun>
    <Urun>Masaüstü</Urun>
    <Urun>Sistem</Urun>
    <Urun>Web</Urun>
</Urunler>
<Urunler Tip="Eğitim">
    <Urun>Android</Urun>
    <Urun>Oracle</Urun>
    <Urun>Java</Urun>
</Urunler>
</MagazaAnket>',
(SELECT Demographics FROM Sales.Store WHERE BusinessEntityID = 292));
```

---

Veri aktarımı tamamlandı. **Magazalar** tablosundaki veriyi görüntüleyelim.

---

```
SELECT * FROM Magazalar;
```

---

	MagazaID	Anket_UnTyped	Anket_Typed
1	292	<MagazaAnket><YillikSatis>145879</YillikSatis><YillikGelir>79277</YillikGelir>...	<StoreSurvey xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure...>

## XML.QUERY

XQuery'nin uygulanmasını sağlar. XML veriye, XQuery sorgularıyla ulaşarak birden fazla parçasının sonuç olarak döndürülmesi için kullanılır. XQuery ile `doc()` ya da `collection()` gibi fonksiyonlarla bir dosyadan ya da bellek üzerindeki bir değişkenden okuma yapılabilir.

### Söz Dizimi:

---

```
nesne.query('XQuery')
```

---

XQuery ile XML içerisindeki tüm ürünleri listeleyelim.

---

```
SELECT Anket_UnTyped.query('/MagazaAnket/Urunler/Urun') AS Urun
FROM Magazalar;
```

---

Sorgu sonucundaki kayıtları görüntüleyebilmek için bağlantı formatındaki XML sonucuna tıklamanız yeterlidir.

---

```
<Urun>Mobil</Urun>
<Urun>Masaüstü</Urun>
<Urun>Sistem</Urun>
<Urun>Web</Urun>
<Urun>Android</Urun>
<Urun>Oracle</Urun>
<Urun>Java</Urun>
```

---

XQuery ile XML içerisindeki mağaza anketlerini listeleyelim.

---

```
SELECT
    Anket_UnTyped.query('/MagazaAnket') AS UnTyped_Info,
    Anket_Typed.query('declare namespace ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey"; /ns:StoreSurvey') AS Typed_Info
FROM Magazalar;
```

---

	UnTyped_Info	Typed_Info
1	<pre>&lt;MagazaAnket&gt;&lt;YillikSatis&gt;145879&lt;/YillikSatis&gt;&lt;YillikGelir&gt;79277&lt;/YillikGelir&gt;&lt;BankaAd&gt;HIC Bank&lt;/BankaAd&gt;...</pre>	<pre>&lt;StoreSurvey xmlns="http://schemas.microsoft.com/sqlservr/2004/07/adventure-works/StoreSurvey"&gt;...</pre>

Mağaza anketlerini listelerken, query ile namespace kullandığımıza dikkat edin. **AdventureWorks** veritabanındaki **StoreSurvey** isimli namespace'i kullandık.

Şimdi de, aynı yöntemi kullanarak sadece **YillikSatis** bilgisini alalım.

---

```
SELECT
    Anket_UnTyped.query('/MagazaAnket/YillikSatis') AS UnTyped_Info,
    Anket_Typed.query('declare namespace ns="http://schemas.
    microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";/
    ns:StoreSurvey/ns:AnnualSales') AS Typed_Info
FROM Magazalar;
```

---

UnTyped_Info	Typed_Info
1 <YillikSatis>145879</YillikSatis>	<ns:AnnualSales xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">800000...

Tüm ürünler, tip kategorili olarak listeleyelim.

---

```
SELECT
    Anket_UnTyped.query('/MagazaAnket/Urunler') AS Urunler
FROM Magazalar;
```

---

Urunler
1 <Urunler Tip="Yazilim"><Urun>Mobil</Urun><Urun>Masaustu</Urun><Urun>Sistem</Urun><Urun>Web</Urun></Urunler><Urunler Tip="Egitim"><Urun>Android</Urun><Urun>Oracle...

Sadece, Yazılım tipindeki ürünleri listeleyelim.

---

```
SELECT
    Anket_UnTyped.query('/MagazaAnket/Urunler[@Tip="Yazilim"]') AS Yazilim
FROM Magazalar;
```

---

Yazilim
1 <Urunler Tip="Yazilim"><Urun>Mobil</Urun><Urun>Masaustu</Urun><Urun>Sistem</Urun><Urun>Web</Urun></Urunler>

Bazen XML içerisinde arama yaparak, aranan metni içeren kayıtlar bulunmak istenebilir. XML içerisindeki ürünlerde 'Mob' metnini içeren kayıtları bulalım.

---

```
SELECT Anket_UnTyped.query(
    for $b in /MagazaAnket/Urunler/Urun
    where contains($b, "Mob")
    return $b
) AS Urun
FROM Magazalar;
```

---

Kullandığımız XML'de bu şartları, sadece Mobil kategorisi sağladığı için tek kayıt bulundu.

	Urun
	1 <Urun>Mobil</Urun>

## XML.EXIST

Özel bir veri tipinin var olup olmadığını test eder. Test edilen XML örneğinde, belirli bir düğüm ya da attribute girişi olup olmadığına bakar. Dışarıdan aldığı ifadenin XML düğümü içerisinde bulunması halinde **true** (1), bulunamaması halinde **false** (0) değerini döndürür.

XML içerisinde, kaç adet Urunler listesinin bulunduğu sorgulayalım.

---

```
SELECT Anket_UnTyped.exist('(/MagazaAnket/Urunler)[2]') AS Sales_UnTyped
FROM Magazalar;
```

---

Sales_UnTyped	
	1

XML'in yapısını incelerseniz, iki adet farklı tiplere sahip **Urunler** listesi bulunur.

```
<Urunler Tip="Yazilim">
  <Urun>Mobil</Urun>
  <Urun>Masaustu</Urun>
  <Urun>Sistem</Urun>
  <Urun>Web</Urun>
</Urunler>
<Urunler Tip="Egitim">
  <Urun>Android</Urun>
  <Urun>Oracle</Urun>
  <Urun>Java</Urun>
</Urunler>
```

Eğer kare parantez içerisinde 3 değerini parametre olarak gönderseydik, sorgu 0 değerini döndürecekてい. Çünkü XML içerisinde 3 farklı **Urunler** listesi yoktur.

Bu **Urunler** isimli iki listenin de içerisinde, farklı sayıda **Urun** bilgisi bulunuyor. Toplamda 7 adet olan **Urun** bilgilerini sorgulayarak doğruluğunu test edelim.

---

```
SELECT
Anket_UnTyped.exist('(/MagazaAnket/Urunler/Urun)[7]') AS Sales_
UnTyped
FROM Magazalar;
```

---

Sales_UnTyped	
	1

Bu sorguda da, kare parantez içerisinde, 7 yerine farklı bir değer kullandık, 0 geri dönüşü olacaktır.

XML üzerinde yapılan bu işlemler, XML verisini değişken ortamına alınarak da yapılabilir.

**IsTuru** bilgisi 'CO' olan **kayıtin** var olup olmadığını sorgulayalım.

---

```
DECLARE @xml XML;
DECLARE @exist BIT;
SET @xml = (SELECT Anket_UnTyped FROM Magazalar);
SET @exist = @xml.exist('/MagazaAnket[IsTuru="CO"]');
SELECT @exist;
```

---

(No column name)	
1	1

Sonuç olarak dönen 1 değeri, 'CO' isimli bir iş türünün bulunduğu gösterir. **IsTuru** bilgisi için farklı bir metinsel değer belirtildiğinde, 0 değeri dönecektir.

**exist** metodu, bir **WHERE**filtresi ile birlikte de kullanılabilir. Örneğin, **Eğitim** ürünleri içerisinde, iş türü 'CO' olan kaydın var olup olmadığını öğrenelim.

---

```
SELECT
Anket_UnTyped.query('/MagazaAnket/Urunler[@Tip="Egitim"]') AS Egitim
FROM Magazalar
WHERE Anket_UnTyped.exist('/MagazaAnket[IsTuru="CO"]') = 1;
```

---

Egitim
<Urunler Tip="Egitim"><Urun>Android</Urun><Urun>Oracle</Urun><Urun>Java</Urun></Urunler>

Burada kullanılan 1 değeri, **exist** metodundan dönmesi beklenen değerin filtrelenmesi için kullanılır. **exist** metodundan 0 değeri dönerse, **Eğitim** tipinde de olsa, ürün bilgisi iş türü olarak 'CO' koşulunu karşılamayacağı için sorgu sonuç döndürmeyecektir.

Ancak, **IsTuru** olarak XML de karşılığı olmayan bir değer verilip, eşitlik olarak da 0 değeri kullanılırsa, bu şartlara uyacak kayıtlar getirilecektir. Çünkü XML'de olmayan kayıtlar sorgulanmış olacaktır.

---

```
SELECT
Anket_UnTyped.query('/MagazaAnket/Urunler[@Tip="Egitim"]') AS Egitim
FROM Magazalar
WHERE Anket_UnTyped.exist('/MagazaAnket[IsTuru="CE"]') = 0;
```

---

Egitim
1 <Urunler Tip="Egitim"><Urun>Android</Urun><Urun>Oracle</Urun><Urun>Java</Urun></Urunler>

## XML.VALUE

Belirli bir element ya da attribute için sınırlı değere erişime izin verir. XQuery'nin sonucunu, bir SQL Server veri tipine dönüştürerek skaler bir sonuç döndürür. Ancak, XQuery ifadesi ile verilen düğüm, bir tek skaler elemana dönüştürülebilir bir düğüm olmalıdır.

**YillikSatis** bilgisinin değerini skaler bir türe dönüştürerek görüntüleyelim.

---

```
SELECT Anket_UnTyped.value('(/MagazaAnket/YillikSatis)[1]', 'INT') AS Satis
FROM Magazalar;
```

---

Satis
1 145879

Önceki örnek gibi satış bilgilerini skaler bir türe dönüştürerek görüntüleyelim.

---

```
SELECT
Anket_UnTyped.value('(/MagazaAnket/YillikSatis)[1]', 'INT') AS Satis,
Anket_Typed.value('declare namespace ns="http://schemas.microsoft.
com/sqlserver/2004/07/adventure-works/StoreSurvey";(/ns:StoreSurvey/
ns:AnnualSales)[1]', 'INT') AS Satis
FROM Magazalar;
```

---

Satis	Satis
1 145879	800000

Ürün tipi 2 olan ürün tipini görüntüleyelim.

---

```
SELECT
Anket_UnTyped.value('(/MagazaAnket/Urunler/@Tip)[2]', 'VARCHAR(10)') AS Tip
FROM Magazalar;
```

---

Tip
1 Egitim

Ürün tipi olarak 1 değeri gönderildiğinde ise **Yazılım** görüntülenecektir.

XML veri içerisindeki uzmanlık alanındaki bilgileri görüntüleyelim. Bu işlemi yaparken de, XML'den alınan bir değeri, **CONCAT** fonksiyonunu kullanarak bir metin ile birleştirerek görüntüleyelim.

---

```
SELECT
Anket_UnTyped.value('concat("Uzmanlık: ", (/MagazaAnket/Uzmanlik)[1])',
'VARCHAR(20)') AS Uzmanlık
FROM Magazalar;
```

---

	Uzmanlık
1	Uzmanlık: Technology

## XML.NODES

XML veriyi, daha fazla ilişkisel biçimde satırlara ayırmayı sağlar. Bir XQuery sorgusu ile yeni bir XML düğümü oluşturur.

Oluşturulan bu XML doğrudan değil, XML metotları ya da **CROSS APPLY** ya da **OUTER APPLY** ile kullanılabilir.

---

```
DECLARE @veritabanlari XML;
SET @veritabanlari =
'<Urunler>
<Urun>SQL Server</Urun>
<Urun>Oracle</Urun>
<Urun>DB2</Urun>
<Urun>PostgreSQL</Urun>
<Urun>MySQL</Urun>
</Urunler>'
SELECT Kategori.query('.//text()') AS VeritabaniTuru
FROM @veritabanlari.nodes('/Urunler/Urun') AS Urun(Kategori);
```

---

	VeritabaniTuru
1	SQL Server
2	Oracle
3	DB2
4	PostgreSQL
5	MySQL

XML içerisindeki **Tip** bilgisine göre bir **CROSS APPLY** sorgusu oluşturarak **Yazılım** tipindeki ürünleri listeleyelim.

---

```
SELECT Kategori.query('.//text()') AS VeritabaniTuru FROM Magazalar
CROSS APPLY Anket_UnTyped.nodes('/MagazaAnket/Urunler[@
Tip="Yazılım"]/Urun') AS Urun(Kategori);
```

---

VeritabaniTuru
1 Mobil
2 Masaüstü
3 Sistem
4 Web

## XML.MODIFY() İLE XML VERİYİ DÜZENLEMEK

XQuery'e veri değişikliği yapabilme yeteneği kazandırır. XML DML olarak da adlandırılabilir. Bir XML veri üzerinde modify ile düzenleme yapılabilir.

Bir XML değişken tanımlayalım. Bu değişkene değer atayalım ve üzerinde değişiklikler yapalım.

---

```
DECLARE @xmlAraba XML;
SET @xmlAraba = '
<arabalar>
<araba></araba>
<araba></araba>
</arabalar>';
SELECT @xmlAraba;

SET @xmlAraba.modify(
'insert attribute renk{"siyah"}
into /arabalar[1]/araba[1]')
SELECT @xmlAraba;
```

---

(No column name)
1 <arabalar><araba /><araba /></arabalar>
(No column name)
1 <arabalar><araba renk="siyah" /><araba /></arabalar>

İlk olarak araba bilgilerinde herhangi bir veri yoktu. Daha sonra, modify ile **INSERT** işlemi gerçekleştirerek renk adında yeni bir alan açtık ve veri olarak da siyah değerini girdik.

**MagazaAnket** örneğimize geri dönerken, modify işlemini bu veri üzerinde gerçekleştirelim.

---

```
UPDATE Magazalar
SET Anket_UnTyped.modify(`
    insert(<Aciklama>Yazılım ve eğitim çözümleri</Aciklama>
    after(/MagazaAnket/CalisanSayisi)[1]'),
    Anket_Typed.modify('declare namespace ns="http://schemas.
    microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";
    insert(<ns:Comments>Yazılım ve eğitim çözümleri</ns:Comments>
    after(/ns:StoreSurvey/ns:NumberEmployees)[1]')
WHERE MagazaID = 292;
```

---

(1 row(s) affected)

Bu sorgu ile birlikte, XML verimiz içerisinde **Aciklama** adında yeni bir alan oluşturdu. Bu alanın namespace içerisindeki karşılığı ise **Comments** alanıdır.

## DELETE

modify metodu kullanılarak XML veri üzerinde **DELETE** işlemi de yapılabilir. Bu işlem için delete ifadesi kullanılır.

Az önce oluşturduğumuz açıklama alanı üzerinde silme işlemi yapalım.

---

```
UPDATE Magazalar
SET Anket_UnTyped.modify('delete(/MagazaAnket/Aciklama)[1]')
WHERE MagazaID = 292;
```

---

**replace value of** ifadesi ile düğüm üzerinde düzenleme yapılabilir. Bu ifade, düğümde güncelleme yapmak için kullanılabilir.

## REPLACE VALUE OF

```
UPDATE Magazalar
SET Anket_UnTyped.modify('replace value of (/MagazaAnket/Aciklama/
text())[1] with "2. açıklama"')
WHERE MagazaID = 292;
```

---

# XML BİÇİMİNDEKİ İLİŞKİSEL VERİYE ERIŞMEK

SQL Server 2005 ile birlikte, XML entegrasyonu ve veri erişim mimarisine birçok farklı özellik eklendi. Bu özelliklerle birlikte, XML ile çalışmak daha kolay ve işlevsel hale geldi.

## FOR XML

Veritabanı tablolarındaki veriyi XML formatı ile istemcilere ulaşımaya hazır hale getirir. Yani, FOR XML deyimi, ilişkisel veriler üzerinden XML veri elde etmek için kullanılır.

### Söz Dizimi:

---

```
SELECT sutun_isimleri
FROM tablo_isim
FOR XML { RAW | AUTO | EXPLICIT | PATH }
```

---

**FOR XML**, çıktı veriyi belirleyecek farklı modlara sahiptir. Bunlar;

- **RAW**: Sonuç kümesindeki her veri satırını, tek bir veri elemanı olarak geri döndürür. Veri elemanı, satır olarak adlandırılır ve her sütun bir satır elemanın attribute'ı olarak listelenir. Satır kontrolü kolay olacağı için iç içe **FOR XML** ifadeleri ile iç içe XML yapıları oluşturulabilir.
- **AUTO**: Her bir elemanı ya tablo ismi ile ya da verinin kaynağı olan tablonun takma adı ile sınıflandırılır. **SELECT** ifadesi sonucunda, tablo yapısını XML'de satır bazında göstererek XML çıktı üretir.
- **EXPLICIT**: XML yapısını etkin olarak biçimlendirmek için kullanılır. **PATH** özelliği geniş oranda bu özelliğin yerini alsa da geçmişe dönük uyumluluk açısından desteklenmektedir.
- **PATH**: **EXPLICIT** özelliğine benzer. XML yapısı üzerinde etkin biçimlendirme için kullanılır. Ancak geliştirmesi ve yönetimi daha kolaydır.

XML biçimlendirme özelliklerine ek olarak, XML sonuçlarını düzenlemek için kullanılabilecek bazı parametreler de vardır.

- **ROOT**: Bu özellik, XML'de bir kök düğüm eklemenizi sağlar. Bu kök düğümü

eklemek zorunlu değildir. Belirttiğiniz kök düğüme isim verebileceğiniz gibi varsayılan olarak **root** ismiyle de kullanabilirsiniz.

- **TYPE:** SQL Server'a, sonuçları varsayılan unicode karakter tipinde değil, XML veri tipinde raporlamasını bildirir.
- **ELEMENTS:** Sonuç verisinde, sütunların attribute'ler halinde değil, içe yerleştirilmiş elemanlar olarak döndürülmesini sağlar. **AUTO** ve **RAW** biçimlendirme özellikleriyle birlikte kullanılır.
- **BINARY BASE 64:** **Image**, **Binary**, **VarBinary** gibi sütunların base64 biçiminde kodlanması sağlar.
- **XML DATA:** Sonuçlara öncelikle bir XML şema uygulamak için kullanılır. Uygulanan bu schema, veriler için, veri yapısı ve kurallar tanımlayacaktır.

## RAW

**ProductID** değerine göre koşul ile sorguladığımız kayıtları XML olarak görüntüleyelim.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML RAW;
```

---

Sorgu sonucunda görüntülenen XML dosyası şu şekilde olacaktır.

---

```
<row ProductID="1" Name="Adjustable Race" ProductNumber="AR-5381" />
<row ProductID="2" Name="Bearing Ball" ProductNumber="BA-8327" />
<row ProductID="3" Name="BB Ball Bearing" ProductNumber="BE-2349" />
<row ProductID="4" Name="Headset Ball Bearings" ProductNumber="BE-2908"/>
```

---

Bu sorguda satır bazlı XML oluşturulması söz konusudur. Ancak XML yapısına uygun bir model değildir. Daha temiz ve kullanışlı bir XML için aşağıdaki sorgu kullanılabilir.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML RAW, ELEMENTS;
```

---

**ELEMENTS** eklendikten sonra oluşturulan sorgunun sonucu şu şekilde olacaktır.

---

```
<row>
  <ProductID>1</ProductID>
  <Name>Adjustable Race</Name>
  <ProductNumber>AR-5381</ProductNumber>
</row>
<row>
  <ProductID>2</ProductID>
  <Name>Bearing Ball</Name>
  <ProductNumber>BA-8327</ProductNumber>
</row>
<row>
  <ProductID>3</ProductID>
  <Name>BB Ball Bearing</Name>
  <ProductNumber>BE-2349</ProductNumber>
</row>
<row>
  <ProductID>4</ProductID>
  <Name>Headset Ball Bearings</Name>
  <ProductNumber>BE-2908</ProductNumber>
</row>
```

---

**<row>** ifadesi hoşunuza gitmediyse bu yapıyı değiştirerek kendi element isminizi belirleyebilirsiniz.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML RAW('kodlab');
```

---

Yukarıdaki soru sonucunda, daha önce **<row>** ile belirtilen sütunların artık **<kodlab>** olarak belirtildiğini görebilirsiniz.

---

```
<kodlab ProductID="1" Name="Adjustable Race" ProductNumber="AR-5381" />
<kodlab ProductID="2" Name="Bearing Ball" ProductNumber="BA-8327" />
<kodlab ProductID="3" Name="BB Ball Bearing" ProductNumber="BE-2349" />
<kodlab ProductID="4" Name="Headset Ball" ProductNumber="BE-2908" />
```

---

Bu XML dokümanına bir root element de eklenebilir.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML RAW('kodlab'), ROOT('DIJIBIL');
```

---

XML'e root element eklendikten sonra görünüm şu şekilde olacaktır.

---

```
<DIJIBIL>
  <kodlab ProductID="1" Name="Adjustable Race" ProductNumber="AR-5381" />
  <kodlab ProductID="2" Name="Bearing Ball" ProductNumber="BA-8327" />
  <kodlab ProductID="3" Name="BB Ball Bearing" ProductNumber="BE-2349" />
  <kodlab ProductID="4" Name="Headset Ball" ProductNumber="BE-2908" />
</DIJIBIL>
```

---

Tüm veri işlemlerinde olduğu gibi XML yapısında da **NULL** veri bazen sorun çıkarabilir. Yukarıda kullanılan yöntemler ile **NULL** değer içeren bir sütun sorgulandığında o sütuna ait herhangi bir kayıt görüntülenemeyecektir.

Ancak yeni özelliklerden biri olan **XSINIL** ifadesi ile **NULL** değer içeren sütunlar da görüntülenebilir.

---

```
SELECT ProductID, Name, ProductNumber, Color
FROM Production.Product
WHERE ProductID < 5
FOR XML RAW, ELEMENTS XSINIL;
```

---

Bu işlem **RAW** ya da **AUTO** ile birlikte, **ELEMENTS** deyiminin sonuna ekleyerek kullanılabilir. **Color** sütunundaki **NULL** kayıtları da görüntüleyecek bu sorgunun sonucu aşağıdaki gibidir.

---

```
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>1</ProductID>
  <Name>Adjustable Race</Name>
  <ProductNumber>AR-5381</ProductNumber>
  <Color xsi:nil="true" />
</row>
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>2</ProductID>
```

```

<Name>Bearing Ball</Name>
<ProductNumber>BA-8327</ProductNumber>
<Color xsi:nil="true" />
</row>
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ProductID>3</ProductID>
    <Name>BB Ball Bearing</Name>
    <ProductNumber>BE-2349</ProductNumber>
    <Color xsi:nil="true" />
</row>
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ProductID>4</ProductID>
    <Name>Headset Ball Bearings</Name>
    <ProductNumber>BE-2908</ProductNumber>
    <Color xsi:nil="true" />
</row>

```

---

## AUTO

Tablodaki her elementi bir tablo satırı olarak oluşturan **AUTO** komutu, ana element olarak şema adı ve tablo isminden (Örn; **Production.Product**) oluşan bir isimlendirme kullanır.

En temel **FOR XML AUTO** kullanımı;

---

```

SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML AUTO;

```

---

Sorgu sonucu olarak şu şekilde bir XML üretilir.

---

```

<Production.Product ProductID="1" Name="Adjus.." ProductNumber="AR-5381" />
<Production.Product ProductID="2" Name="Beari.." ProductNumber="BA-8327" />
<Production.Product ProductID="3" Name="BB Ba.." ProductNumber="BE-2349" />
<Production.Product ProductID="4" Name="Heads.." ProductNumber="BE-2908" />

```

---

\* *Name* sütununun uzun olması nedeniyle içerikler ... ile kısaltılmıştır.

Oluşturulan bu XML'e bir **root** nod'u da eklenebilir.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML AUTO, ROOT('DIJIBIL');
```

---

**root** nod'u eklenmekten sonra üretilen XML çıktı şu şekildedir.

---

```
<DIJIBIL>
<Production.Product ProductID="1" Name="Adjus.." ProductNumber="AR-5381" />
<Production.Product ProductID="2" Name="Beari.." ProductNumber="BA-8327" />
<Production.Product ProductID="3" Name="BB Ball" ProductNumber="BE-2349" />
<Production.Product ProductID="4" Name="Headset" ProductNumber="BE-2908" />
</DIJIBIL>
```

---

Aynı verileri attribute değil de element olarak görüntülemek istersek, **ELEMENTS** ifadesini eklememiz yeterli olacaktır.

---

```
SELECT ProductID, Name, ProductNumber
FROM Production.Product
WHERE ProductID < 5
FOR XML AUTO, ELEMENTS, ROOT('DIJIBIL');
```

---

Sorgu sonucunda şu şekilde bir XML üretilecektir.

---

```
<DIJIBIL>
<Production.Product>
  <ProductID>1</ProductID>
  <Name>Adjustable Race</Name>
  <ProductNumber>AR-5381</ProductNumber>
</Production.Product>
<Production.Product>
  <ProductID>2</ProductID>
  <Name>Bearing Ball</Name>
  <ProductNumber>BA-8327</ProductNumber>
</Production.Product>
<Production.Product>
  <ProductID>3</ProductID>
  <Name>BB Ball Bearing</Name>
```

```

<ProductNumber>BE-2349</ProductNumber>
</Production.Product>
<Production.Product>
  <ProductID>4</ProductID>
  <Name>Headset Ball Bearings</Name>
  <ProductNumber>BE-2908</ProductNumber>
</Production.Product>
</DIJIBIL>
```

---

## EXPLICIT

XML'in esnek kullanımı için gerekli olan bazı durumlar olabilir. Bu tür durumlarda farklı XML formatları elde etmek için kullanılır.

**EXPLICIT** ile XML veri oluşturabilmek için sorgu içerisinde iki temel meta sütun bulundurulmak zorundadır.

Bunlar;

- **TAG**: İlk sütun Integer tipinde bir **TAG** numarası almak ve sütun adı da **TAG** olmak zorundadır.
- **PARENT**: İkinci sütunun adı **PARENT** olmak zorundadır.

**TAG** ve **PARENT** sütunları veriler üzerinde hiyerarşi belirlemek için kullanılır.

Ürünler tablosu üzerinde hazırlanacak bir XML için format belirleyelim.

---

```

SELECT
  TOP 5
  1 AS TAG,
  NULL AS PARENT,
  ProductID AS [Product!1!ProductID],
  Name AS [Product!1!Name!element],
  ProductNumber AS [Product!1!ProductName!element],
  ListPrice AS [Product!1!ListPrice!element]
FROM Production.Product
ORDER BY ProductID
FOR XML EXPLICIT;
```

---

## Sorgu sonucunun XML görüntüsü:

---

```
<Product ProductID="1">
  <Name>Adjustable Race</Name>
  <ProductNumber>AR-5381</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
<Product ProductID="2">
  <Name>Bearing Ball</Name>
  <ProductNumber>BA-8327</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
<Product ProductID="3">
  <Name>BB Ball Bearing</Name>
  <ProductNumber>BE-2349</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
```

---

Sorgunun XML çıktısında **ProductID** bir attribute iken, **Name**, **ProductNumber** ve **ListPrice** sütunları bir element olarak görüntüülendi.

Bunun nedeni; **ProductID** sütunu için formatlama yaparken element komutunu eklememiş olmamızdır.

---

```
ProductID AS [Product!1!ProductID]
```

---

Element komutu eklenen diğer 3 sütun ise, element formatında görüntülenmiştir.

---

```
ProductNumber AS [Product!1!ProductNumber!element]
```

---

Aynı sorguda **ProductID** sütunu için de element tanımlaması yaptıktan sonra, artık XML formatındaki dokümanda **ProductID** sütunu da element formatında görüntülenir.

---

```
SELECT
  TOP 3
  1 AS TAG,
  NULL AS PARENT,
  ProductID AS [Product!1!ProductID!element],
```

```
Name AS [Product!1!Name!element],
ProductNumber AS [Product!1!ProductNumber!element],
ListPrice AS [Product!1!ListPrice!element]
FROM Production.Product
ORDER BY ProductID
FOR XML EXPLICIT;
```

---

### Sorgunun XML çıktısı:

---

```
<Product>
  <ProductID>1</ProductID>
  <Name>Adjustable Race</Name>
  <ProductNumber>AR-5381</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
<Product>
  <ProductID>2</ProductID>
  <Name>Bearing Ball</Name>
  <ProductNumber>BA-8327</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
<Product>
  <ProductID>3</ProductID>
  <Name>BB Ball Bearing</Name>
  <ProductNumber>BE-2349</ProductNumber>
  <ListPrice>0.0000</ListPrice>
</Product>
```

---

**NULL** değer içeren sütunlar üzerinde de **EXPLICIT** kullanılabilir.

**ProductID** sütununu **DESCENDING** olarak sıralayarak XML çıktısı oluşturalım.

---

```
SELECT
  TOP 3
  1 AS TAG,
  NULL AS PARENT,
  ProductID AS [Product!1!ProductID!element],
  Name AS [Product!1!Name!element],
  ProductNumber AS [Product!1!ProductNumber!element],
  ListPrice AS [Product!1!ListPrice!element],
```

```
Color AS [Product!1!Color!elementxsinil]
FROM Production.Product
ORDER BY ProductID DESC
FOR XML EXPLICIT;
```

---

### Sorgunun XML çıktısı:

```
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>1004</ProductID>
  <Name>% 20 indirimli ürün</Name>
  <ProductNumber>SK-9299</ProductNumber>
  <ListPrice>0.0000</ListPrice>
  <Color xsi:nil="true" />
</Product>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>999</ProductID>
  <Name>Road-750 Black, 52</Name>
  <ProductNumber>BK-R19B-52</ProductNumber>
  <ListPrice>619.5637</ListPrice>
  <Color>Black</Color>
</Product>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>998</ProductID>
  <Name>Road-750 Black, 48</Name>
  <ProductNumber>BK-R19B-48</ProductNumber>
  <ListPrice>619.5637</ListPrice>
  <Color>Black</Color>
</Product>
```

---

Sorgu sonucunda dönen 3 kayıttan ilkinde, `color` sütunu `NULL` olduğu için就这样被指定了。

```
<Color xsi:nil="true" />
```

---

Ancak diğer kaytlarda bir değer bulunduğu için renk isimleri metinsel olarak doldürüldü.

## EXPLICIT İLE SÜTUNLARI GİZLEMEK

Bazı durumlarda kayıtları XML'e dönüştürürken bazı sütunların gizlenmesi istenebilir. Bu gibi durumlarda HIDE direktifi kullanılır.

Daha önce kullandığımız sorguda ürün fiyatlarını gizleyelim.

---

```
SELECT
    TOP 3
    1 AS TAG,
    NULL AS PARENT,
    ProductID AS [Product!1!ProductID!element],
    Name AS [Product!1!Name!element],
    ProductNumber AS [Product!1!ProductNumber!element],
    ListPrice AS [Product!1!ListPrice!hide],
    Color AS [Product!1!Color!elementxsinil]
FROM Production.Product
ORDER BY ProductID DESC
FOR XML EXPLICIT
```

---

Sorgu içerisinde **ListPrice** sütunu olmasına rağmen, XML çıktısında bu sütun yer almayacaktır.

---

```
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ProductID>1004</ProductID>
    <Name>% 20 indirimli ürün</Name>
    <ProductNumber>SK-9299</ProductNumber>
    <Color xsi:nil="true" />
</Product>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ProductID>999</ProductID>
    <Name>Road-750 Black, 52</Name>
    <ProductNumber>BK-R19B-52</ProductNumber>
    <Color>Black</Color>
</Product>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ProductID>998</ProductID>
    <Name>Road-750 Black, 48</Name>
    <ProductNumber>BK-R19B-48</ProductNumber>
    <Color>Black</Color>
</Product>
```

---

**EXPLICIT** ile kullanılabilecek diğer direktifler:

- ID, IDREF ve IDREFS
- CDATA
- XML
- XMLTEXT

## PATH

**FOR XML PATH** ifadesi yetenekli ve sade bir XML çıktısı üretme komutudur.

Ana kategori ve alt kategoriler üzerinde bir **JOIN** işlemi gerçekleştirerek **FOR XML PATH** ifadesini kullanalım.

---

```
SELECT pc.ProductCategoryID, pc.Name, psc.ProductSubcategoryID, psc.Name
FROM Production.ProductCategory AS pc
INNER JOIN Production.ProductSubcategory AS psc
ON pc.ProductCategoryID = psc.ProductCategoryID
FOR XML PATH;
```

---

Sorgu sonucunda temiz ve anlaşılır bir XML çıktısı üretilecektir. Üretilen XML içerisindeki ilk üç kayıt;

---

```
<row>
  <ProductCategoryID>1</ProductCategoryID>
  <Name>Bikes</Name>
  <ProductSubcategoryID>1</ProductSubcategoryID>
  <Name>Mountain Bikes</Name>
</row>
<row>
  <ProductCategoryID>1</ProductCategoryID>
  <Name>Bikes</Name>
  <ProductSubcategoryID>2</ProductSubcategoryID>
  <Name>Road Bikes</Name>
</row>
<row>
  <ProductCategoryID>1</ProductCategoryID>
  <Name>Bikes</Name>
  <ProductSubcategoryID>3</ProductSubcategoryID>
  <Name>Touring Bikes</Name>
</row>
```

---

Bu XML yapısına element isimleri verilerek daha anlaşılır hale getirilebilir.

---

```
SELECT pc.ProductCategoryID, pc.Name,
       psc.ProductSubcategoryID, psc.Name
  FROM Production.ProductCategory AS pc
 INNER JOIN Production.ProductSubcategory AS psc
    ON pc.ProductCategoryID = psc.ProductCategoryID
FOR XML PATH('Kategori'), ROOT('Kategoriler');
```

---

Bazı bilgiler element olarak değil, bir elemente öznitelik olarak eklenmek istenebilir. Hazırladığımız örnekte **ProductCategoryID** değerini **Kategori** elementine öznitelik olarak ekleyelim.

---

```
SELECT pc.ProductCategoryID "@KategoriID", pc.Name,
       psc.ProductSubcategoryID, psc.Name
  FROM Production.ProductCategory AS pc
 INNER JOIN Production.ProductSubcategory AS psc
    ON pc.ProductCategoryID = psc.ProductCategoryID
FOR XML PATH('Kategori'), ROOT('Kategoriler');
```

---

Bu sorgunun XML çıktısı şu şekilde olacaktır.

---

```
<Kategoriler>
  <Kategori KategoriID="1">
    <Name>Bikes</Name>
    <ProductSubcategoryID>1</ProductSubcategoryID>
    <Name>Mountain Bikes</Name>
  </Kategori>
  <Kategori KategoriID="1">
    <Name>Bikes</Name>
    <ProductSubcategoryID>2</ProductSubcategoryID>
    <Name>Road Bikes</Name>
  </Kategori>
  <Kategori KategoriID="1">
    <Name>Bikes</Name>
    <ProductSubcategoryID>3</ProductSubcategoryID>
    <Name>Touring Bikes</Name>
  </Kategori>
  <Kategori KategoriID="2">
    <Name>Components</Name>
```

---

```
<ProductSubcategoryID>4</ProductSubcategoryID>
<Name>Handlebars</Name>
</Kategori>
</Kategoriler>
```

---

\* Sorgu sonucunun ilk dört kaydıdır.

## OPEN XML

Dışarıdan alınan veriyi ilişkisel veritabanı ortamında saklamayı sağlar.

Bu işlemi sistem prosedürleri ile gerçekleştirir. Bunlar;

- **sp\_xml\_preadedocment**
- **sp\_xml\_removedocment**

İşlem adımları;

- **sp\_xml\_preadedocment** ile XML veri parse edilir.
- DOM'a yerleşen veri **OPENXML** ile parçalanır ve ilişkisel ortama aktarılır.
- XML işlemleri için kullanılan hafızanın boşaltılmasını sağlar. Hafıza SQL Server tarafından scope bitiminde yapılır. Ancak hafıza yönetimini SQL Server'a bırakmadan yapabilmek gereklidir. Bunun için **sp\_xml\_removedocment** kullanılır.

**Books** isminde bir tablo oluşturalım ve bir XML veriyi, **openXML** ile bu tabloya ekleyelim.

---

```
CREATE TABLE Books
(
    BookID INT,
    Name VARCHAR(50),
    Author VARCHAR(50),
    Technology VARCHAR(30)
);
```

---

XML veriyi tabloya ekleyelim.

---

```
DECLARE @xml_data INT;
DECLARE @xmldoc VARCHAR(1000);

SET @xmldoc =
```

```
'<root>
<book BookID="1" Name="İleri Seviye SQL Server T-SQL"
Technology="SQL Server" Author="Cihan Ozhan" />
<book BookID="2" Name="İleri Seviye Android Programlama"
Technology="Android" Author="Kerim Fırat" />
</root>

EXEC sp_xml_preparedocument @xml_data OUTPUT, @xmldoc;
INSERT INTO Books
SELECT * FROM OpenXML(@xml_data,'/root/book') WITH Books;
EXEC sp_xml_removedocument @xml_data;
```

---

XML veriyi, **Books** tablosuna başarıyla ekledik. Tablodaki veriyi listeleyerek görelim.

---

```
SELECT * FROM Books;
```

---

	BookID	Name	Author	Technology
1	1	İleri Seviye SQL Server T-SQL	Cihan Ozhan	SQL Server
2	2	İleri Seviye Android Programlama	Kerim Fırat	Android

Şimdi de, tabloya veri eklemeden, sadece **SELECT** ile bir XML veriyi okuma işlemi gerçekleştirelim.

---

```
DECLARE @DocHandle INT;
DECLARE @XmlDocument NVARCHAR(1000);
SET @XmlDocument =
N'<root>
<Musteri MusteriID="VINET" IletisimAd="Hakan Aydin">
    <Urun UrunID="10248" MusID="VINET" CalID="8" SipTarih="2013-02-02">
        <UrunDetay UrunID="19" Miktar="7"/>
        <UrunDetay UrunID="25" Miktar="5"/>
    </Urun>
</Musteri>
<Musteri MusteriID="LILAS" IletisimAd="Cansu Aycan">
    <Urun UrunID="10283" MusID="LILAS" CalID="6" SipTarih="2013-02-13">
        <UrunDetay UrunID="13" Miktar="2"/>
    </Urun>
</Musteri>
</root>';
```

```
EXEC sp_xml_preparedocument @DocHandle OUTPUT, @XmlDocument;
SELECT *
FROM OPENXML (@DocHandle, '/root/Musteri',1)
    WITH (MusteriID varchar(10), IletisimAd varchar(20));
EXEC sp_xml_removedocument @DocHandle;
```

	MusteriID	IletisimAd
1	VINET	Hakan Aydin
2	LILAS	Cansu Aycan

**AdventureWorks** veritabanı modelinde olan bir XML'i tüm sütunlarıyla birlikte görüntüleyelim.

```
DECLARE @XmlDokumanIslem INT;
DECLARE @XmlDokuman NVARCHAR(1000);
SET @XmlDokuman = N'
<root>
<Musteri MusteriID="VINET" Iletisim="Hakan Aydin">
    <Siparis SiparisID="10248" MusteriID="VINET" CalisanID="5">
        SiparisTarih="2013-02-02">
        <SiparisDetay UrunID="11" Miktar="12"/>
        <SiparisDetay UrunID="42" Miktar="10"/>
    </Siparis>
</Musteri>
<Musteri MusteriID="LILAS" Iletisim="Cansu Aycan">
    <Siparis SiparisID="10283" MusteriID="LILAS" CalisanID="3">
        SiparisTarih="2013-02-04">
        <SiparisDetay UrunID="72" Miktar="3"/>
    </Siparis>
</Musteri>
</root>';

EXEC sp_xml_preparedocument @XmlDokumanIslem OUTPUT, @XmlDokuman;
SELECT *
FROM OPENXML (@XmlDokumanIslem, '/root/Musteri/Siparis/SiparisDetay', 2)
WITH
(
    SiparisID      INT      '.../@SiparisID',
    MusteriID     VARCHAR(10) '.../@MusteriID',
    SiparisTarih   DATETIME '.../@SiparisTarih',
```

```

        UrunID      INT      '@UrunID',
        Miktar     INT      '@Miktar'
    );
EXEC sp_xml_removedocument @XmlDokumanIslem;

```

SiparisID	MusteriID	SiparisTarih	UrunID	Miktar
1	10248	VINET	2013-02-02 00:00:00.000	11
2	10248	VINET	2013-02-02 00:00:00.000	42
3	10283	LILAS	2013-02-04 00:00:00.000	72

**SELECT** işleminden önce, XML yapı değişkene **SET** edilir. Bu XML, **sp\_xml\_preparedocument** sistem prosedürü ile parse edildikten sonra, **SELECT** sorgusu çalıştırılır. Son olarak, XML'in bulunduğu belleğin boşaltılması için **sp\_xml\_removedocument** sistem prosedürü çalıştırılır ve bellek boşaltılır.

Sorguda kullanılan XML verinin yapısı farklı formatlarda görünecek şekilde değiştirilebilir. XML verinin tamamı okunsa da, **SELECT** sorgusu içerisinde, sadece görünmesi istenen sütunlar seçilerek, belirli sütunların görüntülenmesi sağlanabilir.

## HTTP ENDPOINT'LERİ

Web üzerinden istemcilere ulaşmak, uzaktaki servislere erişmek bilişim dünyasının farklı çözümler ürettiği sorunlardan biridir. Web üzerinden bir servislere erişmek istediğiinde güvenlik nedeniyle Firewall gibi bazı sorunlarla karşılaşılır.

Servis mimarili teknolojilerde bu iletişim ilk olarak port açma yöntemi ile gerçekleştiriliyordu. Açılan port ile web ve servis arasında binary veri alış verisi gerçekleştiriliyor ve teorik anlamda çözüm sağlanmış olurdu. Ancak bir sistem yöneticisi açısından bu durum asla böyle olmamalıdır. Çünkü açılan her port aslında bir güvenlik riskidir. İş tecrübeleriniz arasında profesyonel bir IT departmanı ile birlikte çalışma tecrübesi varsa, bu tür port açma isteklerine IT departmanının vereceği olumsuz yanıtı yaşamış olmalısınız.

Port üzerinden binary alış verisi işleminin güvenlik sorunları oluşturmaması nedeniyle SOAP geliştirildi. **SOAP** (*Simple Object Access Protocol*), HTTP üzerinden istekte bulunduğu için ekstra bir port açma işleminin önüne geçmiştir. HTTP portu her sistemde aynıdır ve 80. portu kullanır.

Yani zaten erişime açık olan bir port üzerinden servis iletişimini sağlanmaktadır. SOAP paketi binary değil, metin demeti içerir.

HTTP Endpoint'ler veritabanına değil, sunucu üzerine kurularak birçok özelliğe sahip olduğu için güvenlik açısından risklidir. Yani, Endpoint kurulu sunucular güvenlik riski altındadır.

## **HTTP ENDPOINT VE GÜVENLİK**

HTTP Endpoint kullanımının bazı güvenlik riskleri taşıdığını belirttiğimizde en aza indirmek ve önlemler almak için bazı özelliklere sahiptir.

- **Connect:** Kullanıcının endpoint üzerindeki veriyi görmesine izin verir. Bir kullanıcıya, endpoint üzerindeki veriyi görme izni vermeden, endpoint'ı yönetme izni verilebilir.
- **View Definition:** Kullanıcının bir endpoint ile ilgili metadata bilgisini görmesine izin verir.
- **Control:** Kullanıcının belirli bir endpoint'ı değiştirmesi ya da kaldırmasına izin verir.

## **HTTP ENDPOINT İLE KULLANILACAK VERİ NESNELERİNİN OLUŞTURULMASI**

Endpoint ile Stored Procedure'leri kullanarak bir servis oluşturacağız. Bu servisi oluşturmadan önce iki tane Stored Procedure oluşturacağız.

Tüm ürünleri listeleyen sproc;

---

```
CREATE PROC sp_Products
AS
SELECT ProductID, Name FROM Production.Product;
```

---

Tüm personelleri listeleyen sproc;

---

```
CREATE PROC sp_Persons
AS
SELECT BusinessEntityID, PersonType, FirstName, LastName
FROM Person.Person;
```

---

Oluşturulan Stored Procedure'ler Endpoint ile servis olarak kullanılabilir hale getirilecektir.

## HTTP ENDPOINT OLUŞTURULMASI VE YÖNETİLMESİ

Endpoint'ler için oluşturulan Stored Procedure'leri kullanarak bir HTTP Endpoint oluşturalım.

---

```
CREATE ENDPOINT EP_AdventureWorks
STATE = STARTED
AS HTTP
(
    PATH = '/AWorks',
    AUTHENTICATION = (INTEGRATED),
    PORTS = (CLEAR),
    SITE = 'localhost'
)
FOR SOAP
(
    WEBMETHOD 'Products',
    (NAME = 'AdventureWorks2012.dbo.sp_Products'),
    WEBMETHOD 'Persons',
    (NAME = 'AdventureWorks2012.dbo.sp_Persons'),
    BATCHES = DISABLED,
    WSDL = DEFAULT,
    DATABASE = 'AdventureWorks2012',
    NAMESPACE = 'http://AdventureWorks/'
)
```

---

HTTP Endpoint'lerin oluşturulabilmesi için `CREATE ENDPOINT` nesne oluşturma komutunun çalışıyor olması gereklidir. Ancak SQL Server Express Edition sürümünde bu komut çalışmaz. Lisanslı sürümlerde bu tür sorunlar yaşamazsınız.

SQL Server Express Edition ile alacağınız hata bildirimleri;

This "CREATE ENDPOINT" statement is not supported on this edition of SQL Server.

Eğer sorgunuz başarılı bir şekilde çalıştı ise artık web servisiniz oluşturulmuş demektir. Daha önce web servis geliştirme ya da kullanma tecrübesine sahipseniz aşağıdaki bağlantı ile ilgili stored procedure'lerin dışarıya açılmış servis modellerini inceleyebilirsiniz.

<http://localhost/AWorks?wsdl>

Yukarıdaki servis ile iki prosedüre de ulaşabilirsiniz. Servis isimlerinden herhangi birini çağrıdığınızda, SQL Server'da tanımlı ve çağrıdığınız servis için oluşturulan Stored Procedure çalışacak ve sonucu döndürecektir.



# SQL SERVER TRANSACTION

16

Veritabanı sunucu, birçok işlemi doğru, eş zamanlı ve performanslı sorgulamak için optimize edebilecek yetenekler ile geliştirilmektedir. Veritabanı, sistem üzerinde belli kaynaklara sahiptir. Ve bu kaynakların sunduğu imkanlar, hız ve işlem gücü kapasitesi ile sınırlıdır. Bu işlem gücü ile gerçekleştirilen sorgular çok kısa sürelerde gerçekleştirilese de, zaman açısından belli bir saniye ya da salise gibi sürelerde gerçekleşir.

Bir sorgu içerisinde kullanılan satırlara başka kullanıcılar ya da sorgular da ulaşmak ve üzerinde değişiklik yapmak isteyebilir. Küçük çaplı uygulamalarda bu tür durumlar sorun oluşturmayabilir. Ancak bankacılık ya da benzeri büyük veri işlemleri gerçekleştiren çok istemcili veritabanı mimarilerinde aynı anda çok yoğun sorgular gerçekleştiriliyor olabilir.

Bir bankanın ülke genelinde 300 şubesи ve bankamatik cihazı olduğunu düşünelim. Aynı anda banka hesabından havale işlemi gerçekleştiren ya da herkesin çok sık kullanacağı benzer işlemler gerçekleştiren kullanıcıların veritabanı sunucuları üzerinde oluşturduğu işlem yükünü tahmin edebilirsiniz.

Aynı anda gerçekleştirilen işlemler için veritabanı sunucusunda bir işlem sırası gerçekleştirilmesi gerektiğini düşünüyor olabilirsiniz. Ve bu konuda doğal olarak haklısınız. Bir ya da birkaç tablo üzerinde veri değiştirme işlemi gerçekleştirileceğe ve değişiklik yapmak isteyen sorgu sayısı dakikada milyonları buluyorsa, belirli kriterler oluşturulması ve işlemlerin belirli iş blokları halinde gerçekleşmesi gereklidir.

Bu bölümde, bir ya da birden fazla yiğini (*batch*) tek bir iş bloğu olarak hazırlamayı inceleyeceğiz. İş bloklarının her bir parçasının çalışmasını garanti edecek sorgular oluşturarak, herhangi bir hata olasılığında, işlemleri geri almayı ya da işlem başarıyla gerçekleştirildiyse, gerekli bilgilerin log ve gerçek veri tablolarına işlenmesini inceleyeceğiz.

## TRANSACTION VE ORTAK ZAMANLILIK

Bazı durumlarda, gerçekleştirilmek istenen sorgu birden fazla işlemin aynı anda yapılmasını gerektirebilir. Aynı anda gerçekleşmesi gereken işlemlerden bir ya da birkaçının gerçekleşmemesi halinde başarılı şekilde gerçekleşen işlemlerin de geri alınabilmesi gereklidir. Bu durum atom olma (*atomicity*), diğer bir deyişle parçalanamaz olma özelliği anlamına gelir.

Bu durumu nesnel olarak anlatmak gerekirse, paketlenmiş bir kutu düşünelim. İçerisinde çok parçalı ve birleştirildiğinde anlamlı bir nesneye dönüsecek yap-boz bulunuyor olsun. Size verilen görev ise, bu yap-boz parçaların hepsini doğru bir şekilde birleştirmenizdir. Eğer birleştirilemezseniz, parçalar üzerinde gerçekleştirdiğiniz birleştirme işlemlerini iptal ederek, tüm parçaları ilk haline geri döndürmenizdir. Bunun nedeni, tüm parçaların tek bir bütünü temsil etmesidir. Bu parçalardan herhangi biri yanlış kullanılırsa bütün kesinlikle doğru olmayacağındır. Aynı şekilde, parçaları tek başına da kullanamazsınız. Çünkü bu parçalar sırasıyla ve doğru birleştirildiğinde bir anlam ifade edecektir.

Örnek verdigimiz paketlenmiş yap-boz iş bloğunun veritabanındaki karşılığı transaction'dır.

Bir transaction bloğu SQL Server tarafından şu şekilde ele alınır:

- Transaction bloğu başlatılır. Transaction içerisindeki tüm iş parçacıkları bir bütün olarak tamamlanmak ya da bütün olarak iptal edilmek zorundadır. 4 işlem parçası içeren bir transaction bloğu içerisinde 3 işlem gerçekleşse de 1 işlemde hata alınırsa başarılı olan diğer 3 işlem geri alınır ve blok geçersiz olur. Transaction bloğu otomatik ya da kullanıcı tarafından başlatılabilir. Transaction'ı başlatmak için **BEGIN TRANSACTION** ya da kısa olarak **BEGIN TRAN** kullanılabilir.
- Transaction bloğu içerisinde yapılan her bir işlem bittiğinde işlemin başarılı olup olmadığına bakılır. İşlem başarısız ise, geri alma (**ROLLBACK**) gerçekleştirilir. İşlem başarılı ise diğer işlemlere devam edilir.

- Transaction, içerisindeki işlemler başarılı olarak tamamlandığında **COMMIT** ile bitirilir. Başarısız olunursa **ROLLBACK** ile geri alma işlemi gerçekleştirilir. **ROLLBACK** kullanıldığında, tüm veriler transaction önceki haline geri döner.

Bir veritabanının yetenekleri veri üzerindeki hakimiyet ve mimarisi ile doğru orantılıdır. Veritabanı, veriler üzerinde değişiklik yaparken **ACID** kuralını sağlamalıdır.

**ACID** (*Atomicity, Consistency, Isolation, Durability*) kurallarını inceleyelim.

## **BÖLÜNEMEZLİK (ATOMICITY)**

Transaction işleminin ana özelliği olarak açıkladığımız bölünemezlik prensibini yansıtır. Bir transaction bloğu yarı kalamaz. Yarım kalan transaction bloğu veri tutarsızlığına neden olur. Ya tüm işlemler gerçekleştirilir, ya da transaction başlangıcına geri döner. Yani transaction'ın gerçekleştirdiği tüm değişiklikler geri alınarak gerçekleşmeden önceki haline döner.

## **TUTARLILIK (CONSISTENCY)**

Bölünemezlik kuralının alt yapısını oluşturduğu bir kuraldır. Transaction veri tutarlılığı sağlamalıdır. Yani bir transaction içerisinde güncelme işlemi gerçekleşse ve ya kalan tüm işlemler de gerçekleşmeli ya da güncelle işlemi de geri alınmalıdır. Bu veri tutarlılığı açısından çok önemlidir.

## **İZOLASYON (ISOLATION)**

Her transaction veritabanı için bir istek paketidir. Bir istek paketi (*transaction*) tarafından gerçekleştirilen değişiklikler tamamlanmadan bir başka transaction tarafından görülememelidir. Her transaction ayrı olarak işlenmelidir. Transaction'ın tüm işlemleri gerçekleştikten sonra bir başka transaction tarafından görülebilir.

## **DAYANIKLILIK (DURABILITY)**

Transaction'lar veri üzerinde karmaşık işlemler gerçekleştirebilir. Bu işlemlerin bütünü güvence altına almak için transaction hatalara karşı dayanıklı olmalıdır. SQL Server'da meydana gelebilecek sistem sorunu, elektrik kesilmesi, işletim sisteminden ya da farklı yazılımlardan kaynaklanabilecek hatalara karşı hazırlıklı ve dayanıklı olmalıdır.

Bilgisayar bilimlerinde imkansız diye bir şey yoktur. Bir sistemin hata vermesi yüzlerce sebebe bağlı olabilir. Örneğin; geçtiğimiz yıllarda büyük bir GSM operatörünün veri merkezinin bulunduğu binaya sel basması sonucu veri kaybı yaşamışlardı. Bu durum olasılığı düşük gibi görünebilir. Ancak bahsi edilen konu önemli veriler olunca, hesaplanması gereken olasılıkların sınırı yoktur. Tüm hata olasılıklarına karşı dayanıklı bir sistem geliştirilmelidir.

## TRANSACTION BLOĞU

SQL Server, diğer tüm büyük veritabanı yönetim sistemlerinde olduğu gibi veri işlemlerini birçok farklı katman ve işlem ile gerçekleştirir. Veritabanı motoru katmanında ve birçok alt servis tabanında gerçekleştirilen bu işlemler, sorguların ve verinin yönetilmesi için farklı iş süreçlerini yönetir.

SQL Server üzerinde herhangi bir veri değiştirme işleminde değişiklik anında veritabanına yansımaz. Üzerinde değişiklik yapılan sayfalar daha önce diskten hafızaya alınmamış ise öncelikle **tampon hafıza** (*buffer cache*) denilen hafıza bölgesine çağrırlar. Hafıza bölgesindeki sayfalar üzerinde değişiklikler gerçekleştirilir ve veritabanına hemen yansıtılmaz. Hafızaya çağrılarak üzerinde değişiklik yapılan sayfalara **kırıcı sayfa** (*dirty page*) denir. Kırıcı sayfalarda tutulan değişiklikler gerekli iş süreçleri tamamlandıktan sonra veritabanına kaydedilir. Kırıcı sayfaların veritabanına kaydedilme işlemine ise **arındırma** (*flushing*) denir.

Veri tutarlılığı açısından ve veri kaybını önlemek için yapılan değişiklikler **.LDF** uzantılı transaction log dosyasına kaydedilir. Bu işlem, verinin veritabanına kaydedilmeden önce yapılması önemlidir. Bu log dosyaları disk üzerinde gerçek verinin bulunduğu veri sayfalarına yazılmadan önce, verinin kaybolmasını önlemek amacıyla kullanılır. Herhangi bir olası sorun karşısında, tutulan bu loglar, kaybolan verinin geri getirilmesini sağlar.

SQL Server log işlemleriyle ilgili yönetimi ve olası durumları bölümün ilerleyen kısımlarında detaylıca işleyeceğiz.

## TRANSACTION İFADELERİNİ ANLAMAK

Transaction yönetimi için kullanılan dört farklı ifade vardır. Bu ifadeler ile transaction başlatılabilir (**BEGIN**), işlemler geri alınabilir (**ROLLBACK**), transaction bitirebilir (**COMMIT**) ya da kayıt noktaları (**SAVE**) oluşturulabilir.

## **TRANSACTION'ı BAŞLATMAK: BEGIN TRAN**

Transaction'ın başlangıcını belirtir. Bu kısımdan sonraki tüm işlemler transaction'ın bir parçasıdır. İşlem sırasında oluşabilecek olası sorunlarda geri alma ya da transaction'ın sonlandırılması gerçekleştirilebilir.

### **Söz Dizimi:**

---

```
BEGIN TRAN[SACTION] [transaction_ismi | @transaction_degiskeni]
```

---

## **TRANSACTION'ı TAMAMLAMAK: COMMIT TRAN**

Transaction'ın tamamlandığını ve gerçekleştirilen transaction işlemlerinin kalıcı olarak veritabanına yansıtılması için kullanılır. Transaction tarafından etkilenen tüm değişiklikler, işlemlerin tamamı gerçekleşmese bile, bu işlemden sonra kalıcı hale gelir.

**COMMIT** işleminden sonra gerçekleşen değişikliklerin geri alınması için, bu işlemleri geri alacak yeni bir transaction oluşturulmalıdır. Örneğin; bir transaction ile, nümerik bir sütun üzerinde 10 birim azaltma işlemi yapıldı ise, bu işlemi geri almak için aynı sütun üzerinde 10 birim artırma işlemi yapacak yeni bir transaction oluşturulmalıdır.

### **Söz Dizimi:**

---

```
COMMIT TRAN[SACTION] [transaction_ismi | @transaction_degiskeni]
```

---

## **TRANSACTION'ı GERİ ALMAK : ROLLBACK TRAN**

Transaction'ın gerçekleştirdiği tüm işlemleri geri almak için kullanılır. Yani, yapılan tüm işlemler transaction'ın başlangıcındaki haline geri döner. Verilerdeki değişikliklerin anında kalıcı olarak veritabanına yansıtılmadığını belirtmiştik. **ROLLBACK** ile gerçekleştirilen tüm işlemler geriye alınarak transaction sonucunun tutarlılığı garanti edilir.

**ROLLBACK** işlemi, oluşturduğunuz transaction mimarisine bağlı olarak, **kayıt noktalarına** (*save points*) geri dönüş için de kullanılabilir.

**Söz Dizimi:**


---

```
ROLLBACK TRAN[SACTION] [transaction_ismi | kayit_noktasi_ismi
| @transaction_degiskeni | @kayit_noktasi_degiskeni]
```

---

**SABİTLEME NOKTALARI: SAVE TRAN**

**ROLLBACK** işlemi transaction'da en başa dönmek için kullanılır. Bazen de belirli bir noktaya kadar gerçekleşen işlemlerin geçerli kalması istenebilir. Bu işlemlerden sonra gerçekleşecek işlemler için **ROLLBACK**'e ihtiyaç duyulabilir. Sabitleme noktaları oluşturulması, transaction içerisinde en başa dönmek yerine, belirlenen bir işlem noktasına dönmek için kullanılır.

**Söz Dizimi:**


---

```
SAVE TRAN[SACTION] [ kayit_noktasi_ismi | @kayit_noktasi_degiskeni ]
```

---

**TRANSACTION OLUSTURMAK**

Transaction işlemleri için en uygun örnekler bankacılık ve gsm operatörlerinin veritabanı işlemleridir. Karmaşık ve bir çok iş parçacığı ile gerçekleşen sayısız işlemden oluşan bu tür uygulamalarda iş bloklarının doğru planlanması ve kullanılması önemlidir.

Bir banka veritabanında kullanıcı hesaplarını tutan ve farklı iki banka hesabı arasında havale işlemini gerçekleştirmek için bir uygulama oluşturalım.

Banka müşterilerinin hesap bilgilerini tutan basit bir tablo oluşturalım.

---

```
CREATE TABLE Accounts (
    AccountID CHAR(10) PRIMARY KEY NOT NULL,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Branch INT,
    Balance MONEY
);
```

---

**Accounts** tablosuna kayıtlar girelim.

---

```
INSERT INTO Accounts
VALUES ('0000065127','Cihan','Özhan', 489, 10000),
       ('0000064219','Kerim','Fırat', 489, 500);
```

---

Eklediğimiz kayıtları listeleyelim.

---

```
SELECT * FROM Accounts;
```

---

	AccountID	FirstName	LastName	Branch	Balance
1	0000064219	Kerim	Fırat	489	500,00
2	0000065127	Cihan	Özhan	489	10000,00

Artık müşteri hesapları arasında havale gerçekleştirebiliriz. Şimdi, havale için gerekli adımları inceleyelim.

Havale işlemi gerçekleştirecek bir prosedür için temel gereksinimler;

- Para gönderecek hesap sahibinin bilgisi.
- Parayı alacak hesap sahibinin bilgisi.
- Havale edilecek paranın miktarı.

İş süreçlerini, güvenlik ve yönetilebilirlik nedeniyle Stored Procedure'ler ile gerçekleştirmek doğru bir çözüm olacaktır.

---

```
CREATE PROC sp_MoneyTransfer(
    @PurchaserID CHAR(10),
    @SenderId CHAR(10),
    @Amount MONEY
)
AS
BEGIN
    BEGIN TRANSACTION
    UPDATE Accounts
    SET Balance = Balance - @Amount
    WHERE AccountID = @SenderId
    IF @@ERROR <> 0
        ROLLBACK
    UPDATE Accounts
    SET Balance = Balance + @Amount
```

```

WHERE AccountID = @PurchaserID
IF @@ERROR <> 0
ROLLBACK
COMMIT
END;

```

---

Cihan ÖZHAN isimli banka müşterisinin hesabında 10.000 birim, Kerim FIRAT isimli müşterinin ise 500 birim parası var.

Cihan ÖZHAN, Kerim FIRAT'a 500 birim para havale etmek istiyor. Oluşturulan prosedür ile havale işlemini gerçekleştirelim.

---

```
EXEC sp_MoneyTransfer '0000064219','0000065127',500;
```

---

Havale işleminden önceki ve sonraki hesap görünümü şu şekildedir.

AccountID	FirstName	LastName	Branch	Balance
1 0000064219	Kerim	Firat	489	1000,00
2 0000065127	Cihan	Özhan	489	9500,00

Hesaplar arası 500 birim havale işlemi başarıyla gerçekleştirildi.

## SABİTLEME NOKTASI OLUŞTURMAK: SAVE TRAN

Yukarıdaki Transaction İfadelerini Anlamak bölümünde açıkladığımız sabitleme noktası işlemi için bir kaç örnek yapacağız.

Öncelikle, **SAVE TRAN** çalışma şeklini en basit haliyle kavrayabilmek için basit bir transaction oluşturalım.

---

```

BEGIN TRANSACTION
SELECT * FROM Accounts;

UPDATE Accounts
SET Balance = 500, Branch = 287
WHERE AccountID = '0000064219';
SELECT * FROM Accounts;

SAVE TRAN save_updateAccount;

DELETE FROM Accounts WHERE AccountID = '0000064219';
SELECT * FROM Accounts;

```

```
ROLLBACK TRAN save_updateAccount;
SELECT * FROM Accounts;
ROLLBACK TRAN;
SELECT * FROM Accounts;
COMMIT TRANSACTION;
```

---

Kerim Fırat isimli müşterinin hesabındaki para üzerinde güncelleme yaparak değiştiriyoruz. Aynı müşterinin şubesini de değiştirerek 287 no'lu şubeye aktarıyoruz. Daha sonra, sabitleme noktasına ve sonrasında da işlemi en başına alacak şekilde **ROLLBACK** yapıyoruz.

Sabitleme noktasına geri dönüş gerçekleştirecek bir prosedür oluşturalım. Bu prosedürde **HumanResources.JobCandidate** tablosundan veri silme işlemi gerçekleştirilmek isteniyor. Ancak iş transaction ve prosedür iş birliğine geldiğinde bir çok hata yönetimi ve **IF . . . ELSE** yapısı devreye giriyor.

---

```
CREATE PROCEDURE SaveTran(@InputCandidateID INT)
AS
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT;
IF @TranCounter > 0
    SAVE TRANSACTION ProcedureSave;
ELSE
BEGIN TRANSACTION;
BEGIN TRY
    DELETE HumanResources.JobCandidate
    WHERE JobCandidateID = @InputCandidateID;
    IF @TranCounter = 0
        COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @TranCounter = 0
        ROLLBACK TRANSACTION;
    ELSE
        IF XACT_STATE() <> -1
            ROLLBACK TRANSACTION ProcedureSave;
    DECLARE @ErrorMessage NVARCHAR(4000);
    DECLARE @ErrorSeverity INT;
    DECLARE @ErrorState INT;
```

```

SELECT @ErrorMessage = ERROR_MESSAGE();
SELECT @ErrorSeverity = ERROR_SEVERITY();
SELECT @ErrorState = ERROR_STATE();
RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH;

```

---

Prosedürü çağırıralım.

---

```
EXEC SaveTran 13;
```

---

## TRY-CATCH İLE TRANSACTION HATASI YAKALAMAK

SQL Server 2005'den önceki sürümlerde de kullanılabilen `@@ERROR` değeri, her seferinde kod blokları içerisinde kontrol edilerek hata durumu değerlendirilir. Bu durum programlama tarafında birçok zorluğu beraberinde getirir. SQL Server 2005 ile birlikte Try-Catch yapısı kullanılarak daha anlaşılır ve yönetimi kolay kod blokları oluşturulabilir.

### Söz Dizimi:

---

```

BEGIN TRY
    BEGIN TRAN
    ...
    COMMIT
END TRY
BEGIN CATCH
    PRINT @@ERROR
    ROLLBACK
END CATCH;

```

---

İlk transaction örneğinde `@@ERROR` deyiği ile oluşturulan prosedürü Try-Catch yapısı ile değiştirelim.

---

```

ALTER PROC sp_MoneyTransfer(
    @PurchaserID CHAR(10),
    @SenderId CHAR(10),
    @Amount MONEY
)

```

```

AS
BEGIN TRY
    BEGIN TRANSACTION
    UPDATE Accounts
    SET Balance = Balance - @Amount
    WHERE AccountID = @SenderID

    UPDATE Accounts
    SET Balance = Balance + @Amount
    WHERE AccountID = @PurchaserID
    COMMIT
END TRY
BEGIN CATCH
    PRINT @@ERROR + ' hatasıoluştugu için havale yapılamadı.'
    ROLLBACK
END CATCH;

```

---

## XACT\_STATE() FONKSİYONU

Transaction durumunu öğrenmek için kullanılır. Bu sistem fonksiyonunun döndürdüğü değerler ve anlamları aşağıdaki gibidir.

- 0 : Açık transaction yok.
- -1 : Doomed (*uncommitable*) transaction var.
- 1 : Açık transaction var.

Prosedürü **xact\_state** fonksiyonuna göre değiştirelim.

---

```

ALTER PROC sp_MoneyTransfer(
    @PurchaserID CHAR(10),
    @SenderId CHAR(10),
    @Amount MONEY
)
AS
BEGIN TRY
    BEGIN TRANSACTION
    UPDATE Accounts
    SET Balance = Balance - @Amount
    WHERE AccountID = @SenderId

```

```

UPDATE Accounts
SET Balance = Balance + @Amount
WHERE AccountID = @PurchaserID
COMMIT
END TRY
BEGIN CATCH
IF(XACT_STATE()) = -1
BEGIN
PRINT @@ERROR + ' kodlu hata gerçekleşti. Havale yapılamadı.'
ROLLBACK TRAN
END
IF(XACT_STATE()) = 1
BEGIN
PRINT @@ERROR + ' kodlu hata gerçekleşti. Ancak transaction başarı ile bitirildi.'
COMMIT TRAN
END
END CATCH;

```

---

## **İÇ İÇE TRANSACTION'LAR (NESTED TRANSACTIONS)**

Transaction'ların iç içe kullanılmasıyla gerçekleşen yönteme denir. Birden fazla transaction bloğunu iç içe kullanarak işlemlerin katmanlarını farklı şekillerde yönetmeye imkan verir.

Teknik olarak mümkün olsa da çok kullanılırken dikkatli olunmalıdır. Ayrıca mecbur kalınmadığı takdirde kullanılması önerilmez. Çünkü transaction zaten yapı olarak karmaşık ve hata olasılıkları olan bir işlemdir. Bu karmaşık işlemleri daha da karmaşık hale getirecek iç içe transaction kullanımı işleri daha da zorlaştırabilir.

### **Söz Dizimi:**

---

```

BEGIN TRANSACTION -- Dış Transaction
BEGIN TRY
BEGIN TRAN A -- İç Transaction
BEGIN TRY

```

```
-- sorğu ifadeleri
COMMIT TRAN A
END TRY
BEGIN CATCH
    ROLLBACK TRAN A
END CATCH
ROLLBACK TRAN -- Tüm işlemleri geri al
END TRY
BEGIN CATCH
    ROLLBACK TRAN
END CATCH
```

---

Normal transaction'larda olduğu gibi iç içe transaction'lar için de transaction'ın tamamlanması, yani kapatılması kaynak tüketimi ve veri tutarlılığı açısından önemlidir. İç içe transaction kullanımında bu önem daha da artmaktadır.

## **ORTAK ZAMANLILIK VE İZOLASYON SEVİYELERİ**

Bir veritabanı yönetim sistemindeki veri kaynağına birden fazla kullanıcının doğrudan erişim sağlayarak veri değiştirme işlemleri yapması mantıksal olarak basit görünse de veri uyumluluğu açısından arka plan işlemlerinde farklı bazı özellikler kullanılır. SQL Server, üzerinde işlem yapılmak istenen veri kaynağındaki veriyi diğer transaction işlemlerine karşı izole etmesi gereklidir. Bunun nedeni; üzerinde değişiklik yapılmak istenen verinin bir başka kullanıcı tarafından aynı anda değiştirilme işlemine tabi tutulabilme olasılığıdır. Bir transaction, tablodaki sütun üzerinde güncelleme işlemi yaparken, bir başka transaction da aynı anda bu veriyi silmek isterse ne olur?

SQL Server transaction ile ilgili izolasyon işlemini iki türlü gerçekleştirebilir.

### **KİLİTLEME (LOCKING)**

Üzerinde işlem yapılan veri kaynağını başka transaction'lara karşı kilitlemek için kullanılır. Bir transaction bitimine kadar, bir satır, bir sayfa ya da bir tablonun tamamı başka transaction'ların erişimine kapatılabilir.

## SATIR VERSİYONLAMA (ROW VERSIONING)

Satır versiyonlama ile izolasyon işleminde SQL Server her bir transaction için ilgili verinin mantıksal bir kopyasını oluşturur. Transaction, bu kopya veri üzerinde işlem gerçekleştirdiği için gerçek veri üzerinde herhangi bir kilitleme yapılmasına gerek yoktur. Her transaction ve mantıksal kopya için bir versiyonlama sistemi kullanır. Satır versiyonlama ile izolasyon yapabilmek için önceden veritabanı seviyesinde ayarlamalar yapmak gereklidir. Bu izolasyon modelinde versiyonlamalar `Tempdb` sistem veritabanında saklandığı için Tempdb veritabanı için özel performans ayarlamaları yapılması gereklidir.

## ORTAK ZAMANLI ERIŞİM ANOMALİLERİ

Kilitleme ve versiyonlama izolasyon yöntemleri, aynı verilerin farklı transaction'lar tarafından değiştirilmesi işleminde ortaya çıkabilecek olası sorunları engellemek için kullanılır.

İzolasyon teknik olarak mümkün olsa da, bazı nedenlerle izolasyonda sızma olur ve veriler aynı anda başka transaction'lar tarafından değiştirilir ya da okunursa bazı sorunlar oluşabilir.

Ortaya çıkan bu sorunlara **Ortak Zamanlı Erişim Anomalileri** denir.

## KAYIP GÜNCELLEME (LOST UPDATE)

Aynı kayıt üzerinde aynı anda birçok transaction işleminin değişiklik yapması durumunda son yapılan değiştirme işleminin dediği olur. Bu durumda son değişiklikten öncekiler kayıptır. Basit gibi görünse de önemli bir takım hataya yol açar.

Sinema için bir bilet almak için online satın alma işlemi gerçekleştirdiniz. İşleminiz bir transaction ile gerçekleştirilmeye başlanır. Aynı anda bir başka kişi de satın alma işlemi gerçekleştirdi ve onun için de bir transaction başlatıldı. Bu işlemler ortak zamanlı gerçekleştiği ve veriler üzerinde bir kilit ya da satır versiyonlama bulunmadığı için iki transaction da tamamlanır. Ancak son tamamlanan transaction işleminin dediği olur. Bu durumda bir bilet iki ya da daha fazla kişi satın almış olabilir. Basit görünen büyük bir hata olduğunu daha net anlamış olmalısınız.

## TEKRARLANAMAYAN OKUMA (NON-REPEATABLE READ)

Bir transaction aynı anda bir başka transaction'ın değiştirdiği veriler üzerinde çalışabilir. Bu nedenle transaction ilk açıldıktan sonra okuduğu veri kümесini tekrar okuyamaz. Bu sorunu engellemek için kayıtları kilitleme yöntemi kullanılmalıdır.

## HAYALET OKUMA (FANTOM READ)

Bir transaction, değişikliklere karşı kilitleme yaptığı için ilk okuyabildiği verileri daha sonra tekrar okuyabiliyor. Ama yeni veri eklenmesine engel olamıyor. Bu durumda başlayan transaction'dan sonra eklenen kayıtların veri tutarlığını riske atmaması için veri ekleme işlemi de kilitlenmelidir.

## KIRLI OKUMA (DIRTY READ)

Aynı kaynağa erişen birden fazla transaction'dan biri, değiştirme işlemini sona erdirmeden önce, diğer transaction'lar tarafından okunan kayıtlar gerçek veriler değildir. Bu veriler, henüz tamamlanmamış bir transaction tarafından değiştirilmiştir. Bir transaction, veri üzerinde değişiklik yaparsa ve bu sırada başka bir transaction da bu verileri okuduktan sonra, değişikliği yapan ilk transaction, yaptığı değişiklikleri geri alırsa, sonraki transaction'ların elde ettiği veriler gerçek veriler olmazlar. Bu sorunu önlemek için, ikinci transaction'ın ilk transaction bitene kadar kirli sayfaları okumasını engellemek gereklidir.

## KİLİTLER

SQL Server'da, aynı veri üzerinde, aynı anda, birden fazla kullanıcı tarafından oluşturulacak ifade ve transaction'lar bazı durumlarda işlem çakışmasına sebep olabilir. Bir veri kaynağı üzerinde, bir transaction tarafından değiştirme işlemi gerçekleştirken, bir başka transaction tarafından aynı veri kaynağı üzerinde farklı bir işlem gerçekleştirilmek istenebilir. Bir transaction tamamlanmadan veri tutarlığı gerçekleştirmeyeceği için veri kaynağını transaction tamamlanana kadar diğer ifade ve transaction'lara erişilemez hale getirmek gereklidir. Bu engelleme işlemi kilitler tarafından gerçekleştirilir.

## KİLİTLENEBİLİR KAYNAKLAR

- **Veritabanı:** Veritabanı tamamen kilitlenir. Bu genellikle veritabanında her şeyin değişmesini sağlayan şema (*schema*) değişikliklerinde meydana gelir.
- **Tablo:** Tablo tamamen kilitlenir. Bu kilit, tablo ile ilgili gerçek veri ve tüm nesneleri içerir.
- **Extent:** Tüm extent kilitlenir. Bir extent, sekiz page'den meydana gelir. Bu sekiz page, üzerindeki indeks page'leri ile veri satırları da etkilenir.
- **Page:** Page üzerindeki tüm veri ve indeks key'leri kilitlenir.
- **Key:** Bir indeksteki key ya da key'ler üzerinde kilitleme yapılır.
- **Satır/RID:** Kilit Satır Tanımlayıcı (*Row Identifier, RID*) üzerinde kilitleme yaparak tüm satırı etkiler.

## KİLİT MODLARI

Kaynakları kilitleme ihtiyacıının bir çok çeşidi olduğu gibi, kilitleme modlarının da bir çok çeşidi vardır. Bazı metodlar birbirile uyumsuz iken bazıları hiyerarşik olarak alt ya da üst seviyede tanımlanarak uyumluluk gösterebilir.

Kilit modlarını inceleyelim.

### PAYLAŞILMIŞ KİLİT (SHARED LOCK)

Shared Lock (*paylaşılmış kilit*), herhangi bir değişiklik yapmadan, sadece veriyi okumanız gerekiğinde kullanılır ve en basit kilit tipidir. Kullanıcıları Dirty Read problemlerinden korur.

### ÖZEL KİLİT (EXCLUSIVE LOCK)

Başka bir kilit varsa Exclusive Lock oluşturulamaz. Ya da Exclusive Lock varsa o nesne üzerinde başka bir kilit oluşturulamaz. Yani, kilitli veri üzerinde aynı anda iki değiştirme(güncelleme, silme vb.) işlemi yapılamaz.

### GÜNCELLEŞTİRME KİLİDİ (UPDATE LOCK)

Update Lock, özel bir yer tutucudur. Gerçekten hangi verinin güncellenmesi gerektiğini belirlemek için, veriyi taradıktan sonra Shared Lock'un özel kilit haline gelmesi anlamına gelir. Update Lock, kilitlenmeye engel olur. Engel olunan şey bir kilit değil, çıkışa girme anlamındaki bir kilitlenmedir.

Update Lock'lar; sadece Shared Lock ve Intent Shared Lock'lar ile uyumludur.

## Amaç Kiliti (Intent Lock)

Veri ve kilitler arasındaki nesne hiyerarşisini çözmek için kullanılır. Örneğin; tablo üzerinde oluşturulan bir kilit varsa tüm tabloyu kilitleyecektir. Satır üzerinde oluşturulan kilit ise satırı kilitler. Bir satır üzerinde kilitde sahip olduğunuzda, bir başkası tablo seviyeli kilit oluşturursa tablodaki satır üzerinde ne tür bir işlem yapılmalıdır? Bu tür durumlarda SQL Server tablo içerisindeki tüm satırlarda kilit aramak yerine tabloda kilit olup olmadığını bakar.

Böyle bir kilitlenmeden dolayı sorun yaşamamak için Intent Lock'lar kullanılır. Intent Lock'lar kendi arasında üçe ayrılır.

- **Paylaşılmış Amaç Kilidi (*Intent Shared Lock*):** Bu kilit türü ile shared lock hiyerarşisi düşük seviyede kurulur. Düşük seviyeden kasit page ya da tablo gibi alt katman üzerinde kurulmasıdır. Yani, bir page üzerinde kurulabilir.
- **Özel Amaç Kilidi (*Intent Exclusive Lock*):** Intent Shared Lock ile aynıdır. Tek farkı, düşük seviye nesne üzerinde özel kilit olmasıdır.
- **Özel Amaç Kilidi İle Paylaşım (*Shared With Intent Exclusive Lock*):** Shared Lock ve Intent Exclusive Lock özelliklerini içerir. Shared Lock, nesne hiyerarşisini düşürmek, Intent Lock ise veriyi değiştirmek amacıyla kurulmuştur.

## Şema Kilitleri (Schema Locks)

Şemalar ile ilgili kilitlerdir ve iki çeşittir.

- **Şema Değişiklik Kilidi (*Schema Modification Lock*):** Nesneye şema değişikliği yapılır. Kilit süresi içinde bu nesne üzerinde CREATE, ALTER ve DROP ifadeleri çalıştırılabilir.
- **Şema Denge Kilidi (*Schema Stability Lock*):** Bu kilidin amacı, nesne üzerinde diğer sorguların aktif kilitleri olduğu için Schema Modification Lock kiliti kurulmasını engellemektir.
- **Bulk Updates Lock:** Verinin paralel yüklenmesine izin verir. Yani, tablo diğer eylemler tarafından kilitlense de birden fazla BULK INSERT ya da BCP işlemi gerçekleştirilebilir.

## OPTİMİZER İPUÇLARI İLE ÖZEL BİR KİLİT TIPI BELİRLEMEK

SQL Server'da kilitleme işlemlerindeki kontrolü ele almak için kullanılır. SQL ifadelerinde tablodan sonra kullanılırlar.

### Söz Dizimi:

---

```
FROM tablo_ismi [AS takma_ismi] [[WITH] (ipucu)]
```

---

### Kullanım Yöntemleri:

---

```
FROM tablo_ismi AS takma_ismi WITH (ipucu)
FROM tablo_ismi AS takma_ismi (ipucu)
FROM tablo_ismi WITH (ipucu)
FROM tablo_ismi (ipucu)
```

---

Bu ipuçları, SQL Server kilit yöneticisine bırakmaksızın ifadeler için kilit tipi belirleyemeyi sağlar. yukarıdaki kullanım yöntemlerinden herhangi biriyle kullanılabilir.

## READCOMMITTED

SQL Server için varsayılan özellikle. `READ_COMMITTED_SNAPSHOT` özelliği aktifse bu kilit uygulanmaz.

### Örnek:

---

```
SELECT * FROM Production.Product (READCOMMITTED);
```

---

## READUNCOMMITTED/NOLOCK

Hiç bir kilit elde edilmez. Dirty Read problemleri dahil bir çok probleme neden olabilir.

### Örnek:

---

```
SELECT * FROM Production.Product (READUNCOMMITTED);
```

---

ya da

---

```
SELECT * FROM Production.Product (NOLOCK);
```

---

## **READCOMMITTEDLOCK**

**READCOMMITTED** ile aynı işi yapar. Tek fark, bir kilit nesne için daha fazla gerekmeliğinde bırakılır.

### **Örnek:**

---

```
SELECT * FROM Production.Product (READCOMMITTEDLOCK);
```

---

## **SERIALIZABLE/HOLDLOCK**

Yüksek izolasyon seviyedir ve veri uyumluluğunu garanti eder. Transaction işleminde kilit kurulduktan sonra **ROLLBACK** ya da **COMMIT** ile transaction sonlanana kadar serbest bırakılmaz.

### **Örnek:**

---

```
SELECT * FROM Production.Product (SERIALIZABLE);
```

---

ya da

---

```
SELECT * FROM Production.Product (HOLDLOCK);
```

---

## **REPEATABLEREAD**

Transaction'da kilit kurulduktan sonra **ROLLBACK** ya da **COMMIT** ile transaction sonlanana kadar serbest bırakılmaz. Ancak, yeni veri eklenebilir.

### **Örnek:**

---

```
SELECT * FROM Production.Product (REPEATABLEREAD);
```

---

## **READPAST**

Kilidin serbest bırakılmasını beklemek yerine **Page**, **Extent** ve tablo kilitleri hariç tüm satır kilitleri atlanır.

**Örnek:**


---

```
SELECT * FROM Production.Product (READPAST);
```

---

**ROWLOCK**

Kilidin başlangıç seviyesini satır seviyesinde olmaya zorlar. Kilit sayısı sistemin kilit eşik değerini aşmışsa, kilidin daha düşük granular seviyelerine yükseltilmesi engellenmez.

**PAGLOCK**

Adından da anlaşılacağı gibi Page seviyeli kilit kullanır.

**Örnek:**


---

```
SELECT * FROM Production.Product (PAGLOCK);
```

---

**TABLOCK**

Tablo kilit uygulanmasını sağlar. Tablo taraması durumlarını hızlandırır.

**Örnek:**


---

```
SELECT * FROM Production.Product (TABLOCK);
```

---

**TABLOCKX**

Adından da anlaşılacağı gibi TABLOCK özelliğine benzer şekilde çalışır. İzolasyon seviyesi ayarlamasına bağlı olarak transaction ya da ifade süresince kullanıcıların tabloya erişmesine izin vermez, tabloyu kilitler.

**Örnek:**


---

```
SELECT * FROM Production.Product (TABLOCKX);
```

---

**UPDLOCK**

Adından da anlaşılacağı gibi Update Lock kullanır. Diğer kullanıcıların Shared Lock'ları elde etmesine izin verir, ancak ifade ya da transaction sonlanana kadar veri değişikliğine izin vermez.

**Örnek:**


---

```
SELECT * FROM Production.Product (UPDLOCK);
```

---

**XLOCK**

Kilit granularity seviyesinin nasıl belirlendiğine bakmaksızın özel kilit belirleyebilmeyi sağlar.

**Örnek:**


---

```
SELECT * FROM Production.Product (XLOCK);
```

---

Bu ipuçları, sadece tek satırlık ve tek tabloluk ifadelerde değil, aynı zamanda **JOIN** işlemlerinde bir tablo üzerinde kurulmak için de kullanılabilir.

**Örnek:**


---

```
SELECT * FROM tablo_ismil (ipucu)
JOIN tablo_ismi2 AS ti2
ON tablo_ismil.sutunID1 = ti2.sutunID1;
```

---

**ISOLATION SEVİYESİNİN AYARLANMASI**

Transaction işlemlerinde, aynı anda veri kaynaklarına erişen birden fazla transaction'ın bir diğerinin kaynak erişimlerinden ve değişirdiği verilerden izole edilmesi ile ilgili ayarlardır.

**READ COMMITTED**

SQL Server'da varsayılan seçenektedir. Veri okurken kirli okumayı önler. Ancak transaction bitmeden başka bir transaction tarafından veri değişikliği yapılabilir.

**READ COMMITTED** ile varsayılan bu seviyenin korunması yoluyla **Dirty Read** problemlerini önleyecek yeterli doğruluğa sahip olunduğundan emin olunabilir. Bununla birlikte, **Non-Repeatable Read** ve **Phantom** problemleri oluşabilir.

## **READ UNCOMMITTED**

İzolasyon seviyesi 0 (sıfır), yani hiç bir izolasyon yoktur. Transaction açıkken başka transaction'da veriler üzerinde değişiklik yapabilir. Izolasyon seviyesini **READ UNCOMMITTED** olarak ayarlamak, SQL Server'a herhangi bir kilit oluşturmamasını söyler. Bu izolasyon seviyesi ile çoğu Dirty Read olmak üzere bir çok uyumluluk sorunu ile karşılaşılabilir.

**READ UNCOMMITTED** veri doğruluğu açısından uygun bir izolasyon seçimi değildir. Kullanım amacı da bu tür durumlardır. Bir raporlama işlemi bazen epeyce uzun sürebilir. Bu süre içerisinde tablo ya da satırlarda kilitler oluşturacak izolasyon seviyeleri kullanmak diğer transaction'lar tarafından verilerin okunamaması ya da değiştirilememesi anlamına gelir. Rapor işlemlerinde, genellikle veriler için yüksek veri tutarlılığı ihtiyaç yoktur. Yaklaşık ve orantısal bir işlem için, yeteri kadar tutarlı veri kullanılması gereklidir. Uzun sürecek raporlama işlemlerinde bu izolasyon seviyesinin kullanılması önerilebilir.

## **REPEATABLE READ**

Transaction içerisindeki sorguda geçen tüm veriler kilitlemeye alınır. Kirli hafızanın okunmasına izin vermez.

İzolasyon seviyesini yükseltten **REPEATABLE READ** ile sadece Dirty Read değil, Non-Repeatable Read durumları da engelleyerek ek seviyede uyumluluk koruması sağlar.

## **SERIALIZABLE**

Bir transaction sonlanmadan, aynı anda başka bir transaction ortak kaynaklara erişemez. Lost Update dışında tüm uyumluluk problemlerini engeller. En katı izolasyon seviyesidir.

Bu izolasyon seviyesinde, kullanıcı transaction ile ilgi bir işlem yapıyorsa, başka bir transaction işlemi gerçekleştirmek için bu transaction'ın tamamlanmasını beklemek zorundadır.

## **SNAPSHOT**

Bir transaction başladığında veritabanının o anki kopyasını alır. Snapshot üzerinde satır versiyonlama yapılabilir. Her ifade çalıştırılırken, o anki gerçek veri alındıktan sonra üzerinde işlem yapılması sağlanabilir.

# İZOLASYON SEVİYESİ YÖNETİMİ

SQL Server'da izolasyon işlemleri için aşağıdaki söz dizimi kullanılır.

## Söz Dizimi:

---

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED
| REPEATABLE READ | SNAPSHOT | SERIALIZABLE }
```

---

SQL Server, izolasyon seviyelerini oturum bazında ayarlamak için destek sağlar.

Oturum ile ilgili tüm parametrelerin listesini almak için;

---

```
DBCC USEROPTIONS;
```

---

Yukarıdaki sorgu sonucundaki izolasyon seviyesi ile ilgili olan kısım;

	Set Option	Value
1	textsize	2147483647
2	language	us_english
3	dateformat	mdy
4	datefirst	7
5	lock_timeout	-1
6	quoted_identifier	SET
7	arithabort	SET
8	ansi_null_dflt_on	SET
9	ansi_warnings	SET
10	ansi_padding	SET
11	ansi_nulls	SET
12	concat_null_yi...	SET
13	isolation level	read committed

Oturumdaki izolasyon seviyesini değiştirmek için aşağıdaki ifade kullanılabilir.

## Söz Dizimi:

---

```
SET TRANSACTION ISOLATION LEVEL izolasyon_seviyesi
```

---

İzolasyon seviyesini anlayabilmek için bir güncelleme işlemi gerçekleştirelim.

Ürün fiyatlarında %7'lük bir zam yapacağız.

Öncelikle kayıtların ilk halini görelim.

---

```
SELECT Name, ListPrice FROM Production.Product
ORDER BY ListPrice DESC;
```

---

	Name	ListPrice
1	Road-150 Red, 62	3828,7489
2	Road-150 Red, 44	3828,7489
3	Road-150 Red, 48	3828,7489
4	Road-150 Red, 52	3828,7489
5	Road-150 Red, 56	3828,7489
6	Mountain-100 Silver, 38	3637,9893
7	Mountain-100 Silver, 42	3637,9893

Daha sonra, güncelleme yapmak için bir transaction oluşturalım.

---

```
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 1.07
```

---

Transaction açıldı ve güncelleme işlemi gerçekleşti. Yeni bir sorgu ekranı açalım ve güncellenen veriyi kontrol etmek için veri listeleme sorgusunu bu yeni sorgu ekranında yazarak çalıştırıralım.

---

```
SELECT Name, ListPrice FROM Production.Product
ORDER BY ListPrice DESC;
```

---

Bu **SELECT** sorgusu sonucunu göremeyeceksiniz. Çünkü işlem sonlanmayacaktır. Bunun nedeni, SQL Server varsayılan izolasyon seviyesidir. Bu izolasyon seviyesi hatırlarsanız kirli hafızaya erişmenize engel olacak şekilde tasarlanmıştır.

İzolasyon seviyesini değiştirek aynı işlemi gerçekleştirebiliriz. Oturum kirli hafızadan okumaya izin veren **READ UNCOMMITTED** izolasyon seviyesine göre ayarlandığında değişen veri listelenebilir.

---

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT Name, ListPrice FROM Production.Product ORDER BY ListPrice DESC;
```

---

	Name	ListPrice
1	Road-150 Red, 62	3828,7489
2	Road-150 Red, 44	3828,7489
3	Road-150 Red, 48	3828,7489
4	Road-150 Red, 52	3828,7489
5	Road-150 Red, 56	3828,7489
6	Mountain-100 Silver, 38	3637,9893
7	Mountain-100 Silver, 42	3637,9893

Aynı işlem, transaction seviyeli gerçekleştirmek yerine tek sorgu için de oluşturulabilir.

---

```
SELECT Name, ListPrice FROM Production.Product (NOLOCK)
ORDER BY ListPrice DESC;
```

---

	Name	ListPrice
1	Road-150 Red, 62	3578,27
2	Road-150 Red, 44	3578,27
3	Road-150 Red, 48	3578,27
4	Road-150 Red, 52	3578,27
5	Road-150 Red, 56	3578,27
6	Mountain-100 Silver, 38	3399,99
7	Mountain-100 Silver, 42	3399,99

Sonuçlar istediği gibi kilitten bağımsız olarak listelendi. Oluşturduğumuz güncelleme işlemini geri alarak tekrar veri listelemesi gerçekleştirilelim. **ROLLBACK** komutunu **BEGIN TRANSACTION** ile transaction'ı başlattığınız sorgu ekranında çalıştırmalısınız.

---

ROLLBACK

---

Güncelleme yapılarak kirli hafızada bulunan değişikliklerin gerçek veriye etki etmeden geri alındığını ve önceki verinin gösterildiği görülecektir.

---

```
SELECT Name, ListPrice FROM Production.Product ORDER BY ListPrice DESC;
```

---

	ProductID	Name
1	1	Adjustable Race
2	879	All-Purpose Bike Stand
3	712	AWC Logo Cap
4	3	BB Ball Bearing
5	2	Bearing Ball
6	877	Bike Wash - Dissolver
7	316	Blade

---

**READ COMMITTED** izolasyon seviyesine geçmek için;

---

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

---

**SERIALIZABLE** izolasyon seviyesine geçmek için;

---

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

---

## TRANSACTION BAZLI SNAPSHOT İZOLASYON

Transaction işleminde verinin bir kopyasının oluşturulması sağlanabilir. Transaction'ın snapshot'ı, yani kopyası alınır ve transaction kapanıncaya kadar saklanır. Verilerin az değişiklikle (`UPDATE`) ugradığı, ama çok görüntüleme (`SELECT`) durumlarda kullanılabilir.

Snapshot izolasyon seviyesini kullanmak için veritabanı bazında aşağıdaki ayar yapılmalıdır.

---

```
USE MASTER
ALTER DATABASE AdventureWorks
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

---

Gerekli ayarlamalardan sonra Snapshot izolasyon ile bir transaction gerçekleştirmek için şu genel ifadeler kullanılabilir.

---

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRAN
    SELECT ProductID, Name FROM Production.Product;
COMMIT TRAN
```

---

	ProductID	Name
1	1	Adjustable Race
2	879	All-Purpose Bike Stand
3	712	AWC Logo Cap
4	3	BB Ball Bearing
5	2	Bearing Ball
6	877	Bike Wash - Dissolver
7	316	Blade

## İFADE BAZLI SNAPSHOT İZOLASYON

İfade bazlı snapshot izolasyon, çalıştırılan her ifade için ayrı kopya (*snapshot*) alır. Bu izolasyon yöntemi ile, devam eden transaction içerisinde, her ifade için yeni bir snapshot almasından dolayı verinin son halini transaction'da kullanmayı sağlar.

İfade bazlı snapshot izolasyonu kullanabilmek için;

---

```
USE MASTER
ALTER DATABASE AdventureWorks
SET READ_COMMITTED_SNAPSHOT ON
```

---

Yapılan bu değişiklik ile birlikte, artık **READ COMMITTED** izolasyon seviyesi ile başlatılan transaction'lar için versiyonlama yapmaya başlayacaktır.

## KİLİTLENMELERİ YÖNETMEK

SQL Server'da izolasyon gerçekleştirirken kilitleme seçeneği büyük öneme sahiptir. Kilitlerin yönetimi, çalışma prensipleri, veritabanındaki kilitli nesneler, kilitlerin takip edilmesi ve görüntülenmesi, sistem tarafından kilit mekanizmalarının incelenmesi gibi konuları bu bölümde ele alacağız.

## KİLİTLEMELERİ GÖZLEMLEMEK

Sistemdeki kilitlemeleri kimlerin yaptığını takip etmek için **sp\_who** sistem prosedürü kullanılır.

---

```
sp_who ['kullanici_ismi']
```

---

Sistemdeki tüm kullanıcılar için kilitleme bilgilerini listeleyelim.

---

```
sp_who
```

---

	spid	ecid	status	loginame	hostname	blk	dbname	cmd	request_id
1	1	0	background	sa		0	NULL	LOG WRITER	0
2	2	0	background	sa		0	NULL	RECOVERY WRITER	0
3	3	0	background	sa		0	NULL	LAZY WRITER	0
4	4	0	background	sa		0	master	SIGNAL HANDLER	0
5	5	0	background	sa		0	NULL	LOCK MONITOR	0
6	6	0	background	sa		0	NULL	XE DISPATCHER	0
7	7	0	background	sa		0	NULL	XE TIMER	0

Parametresiz kullanılabilen bu sistem prosedürü ile tüm kilitlemeler listelenebileceği gibi bir kullanıcı ismi verilerek, sadece o kullanıcının kilitlemeleri de listelenebilir.

```
sp_who 'sa'
```

	spid	ecid	status	loginame	hostname	blk	dbname	cmd	request_id
1	1	0	background	sa		0	NULL	LOG WRITER	0
2	2	0	background	sa		0	NULL	RECOVERY WRITER	0
3	3	0	background	sa		0	NULL	LAZY WRITER	0
4	4	0	background	sa		0	master	SIGNAL HANDLER	0
5	5	0	background	sa		0	NULL	LOCK MONITOR	0
6	6	0	background	sa		0	NULL	XE DISPATCHER	0
7	7	0	background	sa		0	NULL	XE TIMER	0

**sp\_who** sistem prosedürü ile tüm kullanıcılar için hangi kullanıcının hangi kaynaklara eriştiğini ve durumlarını görebiliriz.

**sp\_who** sistem prosedürünün çalışması sonucu 9 sütunlu bir kayıt listelenir. Bu sütunların isimleri ve anlamları sırası ile şöyledir.

- **spid:** Sistem işlem (process) ID değeri.
- **ecid:** Çalıştırma içerik değeri.
- **status:** İşlemin durumu.
- **loginame:** İşlemin sahibi olan kullanıcının adı.
- **hostname:** İşlemin sahibi bilgisayarın adı.
- **blk:** İşlemi kilitleyen bir başka işlem varsa o işlemin ID değeri. Eğer yok ise 0'dır.
- **dbname:** İşlemin kullandığı veritabanının adı.
- **cmd:** Çalıştırılan SQL ifadesi ya da SQL Server motoru işlem adı.
- **request\_id:** Özel nedenlerden dolayı çalışan işlemlerin istek numaralarını verir. Yok ise, değer 0'dır.

**sp\_who** gibi **sp\_who2** sistem prosedürü de kullanılabilir.

SPID	Status	Login	HostName	BkBy	DBName	Command	CPUTime	DiskIO	LastBatch	ProgramName	SPID	REQUESTID
1	1	BACKGROUND	sa	.	NULL	LOG WRITER	15	0	01/30 11:27:27		1	0
2	2	BACKGROUND	sa	.	NULL	RECOVERY WRITER	15	0	01/30 11:27:27		2	0
3	3	BACKGROUND	sa	.	NULL	LAZY WRITER	390	0	01/30 11:27:27		3	0
4	4	BACKGROUND	sa	.	master	SIGNAL HANDLER	0	0	01/30 11:27:27		4	0
5	5	BACKGROUND	sa	.	NULL	LOCK MONITOR	0	0	01/30 11:27:27		5	0
6	6	BACKGROUND	sa	.	NULL	XE DISPATCHER	0	0	01/30 11:27:27		6	0
7	7	BACKGROUND	sa	.	NULL	XE TIMER	46	0	01/30 11:27:27		7	0

**sp\_lock** sistem prosedürü kilitli olan kaynakları görüntülemek için kullanılır. Hangi kaynakların, hangi kullanıcılar tarafından kilitlendiği görülebilir.

### Söz Dizimi:

---

```
sp_lock [@spid1 = 'spid1'][,@spid2='spid2']
```

---

**sp\_lock** sistem prosedürü dışarıdan iki parametre alabiliyor olsa da, parametresiz olarak en basit haliyle kullanılabilir.

---

```
sp_lock
```

---

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	7	0	0	DB		S	GRANT
2	53	5	0	0	DB		S	GRANT
3	55	2	5	0	TAB		IX	GRANT
4	55	2	7	0	TAB		IX	GRANT
5	55	7	0	0	DB		S	GRANT
6	55	2	3	0	TAB		IX	GRANT
7	55	2	34	0	TAB		IX	GRANT

Parametreli kullanımı da kolaydır.

- **Tek parametreli sp\_lock:** Bir işlem ID (**processesID**) değeri alır. ID değerini aldığı işlem tarafından gerçekleştirilen kilitlenmeleri gösterir.
- **Çift parametreli sp\_lock:** İki işlem ID (**processesID**) değeri alır. Aldığı iki ID değeri tarafından ortak gerçekleştirilen kilitlenmeleri gösterir.

**sp\_lock** sonucunda listelenen sütunların anlamları;

- **spid:** Sistem işlem (**process**) ID değeri.
- **dbid:** Kilitlenme isteyen veritabanının sistem ID'si.
- **ObjId:** Kilitlemede bulunmak isteyen nesnenin sistem ID'si.
- **IndId:** İndeks ID'si.
- **Type:** Kilitlemenin tipi.
- **Resource:** Kilitlenen kaynak.
- **Mode:** Kilitleme isteyenin kilitleme modu. (**Shared, Exclusive vb.**).
- **Status:** Kilitleme istek durumu (**Granted, Convert, Waiting vb.**).

Uygulamalarda `sp_lock` kullanmanız tavsiye edilmez. Microsoft, SQL Server'ın ilerleyen sürümlerinde `sp_lock` yerine, `sys.dm_tran_locks` sistem katalog view'i kullanmayı planlamaktadır.

## ZAMAN AŞIMINI AYARLAMAK

SQL Server'da fark ettiyseniz transaction'ların çalışma süresi sınırsızdır. `BEGIN TRAN` ile başlattığınız transaction, siz `COMMIT` ya da `ROLLBACK` yapana kadar devam edecektir. Bu durum sistem kaynaklarının sürekli tükenmesi anlamına gelir.

Transaction'ların kaynaklar üzerinde bir süreliğine kilitleme yapabilmelerini sağlamak için `LOCK_TIMEOUT` oturum parametresi kullanılır. Parametre olarak milisaniye alan bu oturum parametresi ile transaction zaman aşımı süresini ve dolayısıyla sistem kaynaklarının performans açısından düzenlenmesini sağlayabilirsiniz.

Varsayılan olarak sınırsız olan zaman aşımı süresinin SQL Server tarafından ayar karşılığı -1'dir.

---

```
SELECT @@LOCK_TIMEOUT
```

---

	(No column name)
1	-1

Bu ayarı değiştirmek ise, `LOCK_TIMEOUT` oturum parametresini `SET` etmek gerekir.

Zaman aşımı süresini on bin milisaniye yapalım.

---



---

```
SET LOCK_TIMEOUT 10000
```

---

Parametre değerine tekrar bakalım.

---

```
SELECT @@LOCK_TIMEOUT
```

---

	(No column name)
1	10000

Zaman aşımı süresini tekrar sınırsız yapmak için -1 değeri ile `SET` etmeniz yeterlidir.

---



---

```
SET LOCK_TIMEOUT -1
```

---

`LOCK_TIMEOUT` değerini tekrar kontrol edelim.

---

```
SELECT @@LOCK_TIMEOUT      15.PNG
```

---

	(No column name)
1	-1

## KİLİTLEME ÇIKMAZI: DEADLOCK

Deadlock (*ölüm kilitlenmesi*), kullanıcıların işlem yaparken, karşılıklı olarak birbirlerinin bir sonraki işlemlerini engelleyen durumlar için kullanılan terimdir. SQL Server böyle bir durumu fark ettiğinde 1205 numaralı mesaj kodunu kullanılır. SQL Server böyle bir deadlock durumunda rastlantısal olarak bir kurban seçer ve transaction'ı `ROLLBACK` ile iptal ederek tüm işlemleri geri alır.

Bir geliştirici olarak bu rastlantıslığa müdahale ederek kurban seçiminde söz sahibi olunabilir.

Bu işlem için `DEADLOCK_PRIORITY` parametresi kullanılır.

---

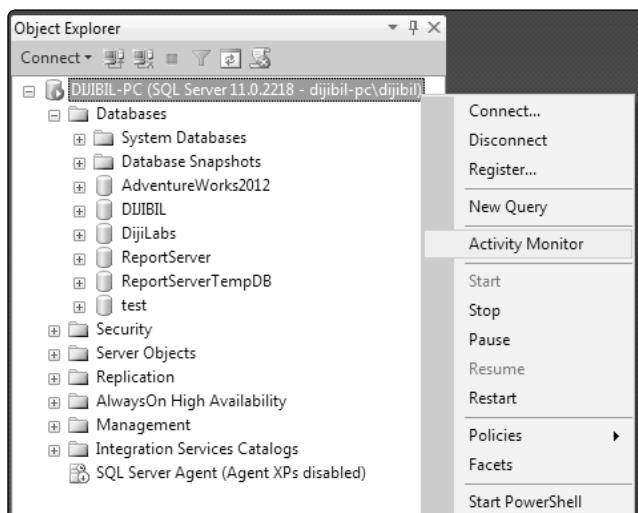
```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority>
@deadlock_var | @deadlock_intvar }
<numeric-priority> ::= { -10|-9|-8|...|0|...|8|9|10 }
```

---

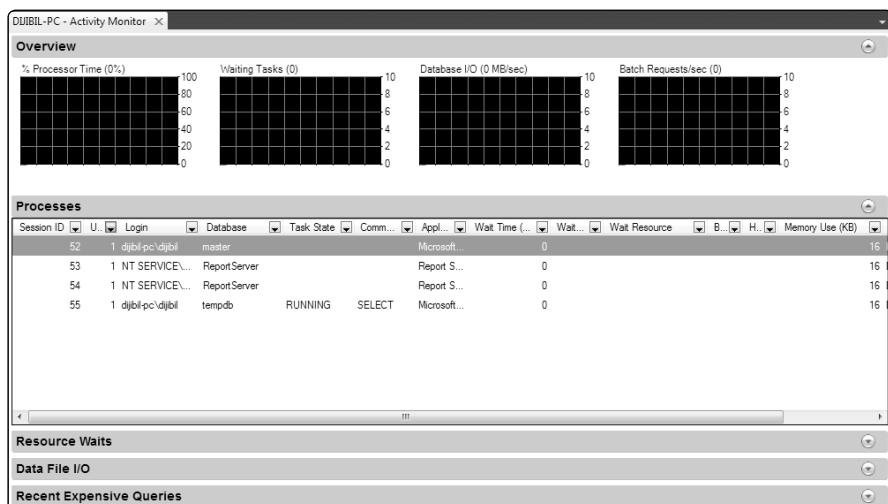
`LOW`, `NORMAL` ya da `HIGH` yerine -10 ile 10 arasında bir değer de atanabilir.

## AKTİVİTE MONİTÖRÜ İLE KİLİTLENMELERİ TAKİP ETMEK VE PROCESS ÖLDÜRMEK

SSMS ile kilitlenmeleri takip etmek için, öncelikle **Aktivite Monitör** açalım.



Resimdeki menüye girdiğinizde aşağıdaki gibi bir ekran ile karşılaşacaksınız.



Ekranda parametreleri inceledikten sonra aşağıdaki sorguları çalıştıralım.

Bu işlemler için iki sorgu ekranı kullanacağız. İlk sorgu ekranında;

---

```
BEGIN TRAN
UPDATE Accounts
SET Balance = Balance - 500
WHERE AccountID = '0000065127';
```

---

İkinci sorgu ekranında;

---

```
SELECT * FROM Accounts;
```

---

Sorgular çalıştırıldıktan sonra aktivite monitörüne tekrar bakıldığından, bir kilitlenme gerçekleştiği görülecektir.

56	1 dijibil\pc\dijibil	DIJİBİL	SUSPENDED	SELECT	Microsoft...	34941 LCK_M_S	keylock hobtid=7205...	51	16
----	----------------------	---------	-----------	--------	--------------	---------------	------------------------	----	----

Şuan kilitlenmiş bir sorgu ekranındaysanız, yeni bir sorgu ekranı oluşturun ve aşağıdaki **İşlem Öldürme** (Process Kill) komutunu kullanarak belirlediğiniz **İşlem ID** değerindeki işlemi sonlandırın.

---

```
KILL 56
```

---

Trigger'lar (*tetikleyiciler*), veri ya da sistemle ilgili değişimlerde otomatik olarak tetiklenen Stored Procedure'lerdir. Trigger'ların Stored Procedure'lerden farkı; dışarıdan parametre almaması, dışarıya parametre göndermemesi ve bir kullanıcı tarafından değil, bir olay tarafından tetiklenmesidir.

## TRIGGER'LARI ANLAMAK

SQL Server'da verilerin yönetiminde neredeyse her şeyi geliştirici belirler ve yönetir. Bu şekilde verileri değiştirmek, silmek ve yeni veriler eklemek gibi işlemler gerçekleştirilir. Ancak bazen belirli işlemlerin otomatik olarak gerçekleştirilmesi gereklidir. Örneğin; bir veri eklendiğinde, aynı ya da farklı bir tabloda, başka bir sütunun değerini değiştirmek gerekebilir. Bu tür durumlarda SQL Server'ın geliştirici tarafından çalıştırılan nesneler yerine, olay olduğu anda tetiklenerek otomatik olarak çalışacak nesnelere ihtiyaç duyulur. Bu tanım, nesne yönelimli programlama tecrübesi olan T-SQL geliştiricileri tarafından çabuk kavranabilecektir. Çünkü nesne yönelimli programlamada event (*olay*) kavramının veritabanı programlamadaki karşılığı trigger'lardır. Veritabanında bir işlem tarafından tetiklenmeler gerçekleştirmek için trigger'lar kullanılır.

Trigger'lar sadece `INSERT`, `UPDATE`, `DELETE` işlemlerinden sonra devreye girebilecek şekilde programlanabilirler. Çoklu tablo ilişkisi (*Inner Join*) bulunan view'lere veri ekleme, sadece trigger'lar ile mümkün olabilir. Aynı zamanda, `INSERT`, `UPDATE`, `DELETE` sorguları çalıştırıldığında, bu işlemlerin yerine yapılmak istenen farklı işlemler de trigger'lar ile gerçekleştirilebilir.

Trigger anlatımında sürekli işlemlerden sonra gerçekleşecek şekilde programlanabileceğinden bahsettik. Çünkü SQL Server, işlemlerden önce çalışması için kullanılan **BEFORE** Trigger özelliğini desteklemez. SQL Server'da olaydan önce devreye giren trigger'lar olarak **INSTEAD OF** kullanılabilir. **INSTEAD OF** trigger'lar, gerçek tablolar değişiklikten etkilenmeden önce devreye girerler.



Oracle eğitimlerimde **BEFORE** Trigger'ları anlatmıştım. Bu trigger'lar bir işlem gerçekleştirilmeden önce çalışırlar. SQL Server bu sorunu **INSERTED** ve **DELETED** adında iki sözde (*pseudo*) tablo ile çözmüştür. Bu tabloları ıllerleyen böülümlerde inceleyeceğiz. Oracle'in **BEFORE** Trigger'i ile SQL Server'in **INSTEAD OF** ve **INSERTED**, **DELETED** tablo modeli açısından sonuç anlamında pek fark yoktur. Ancak kullanılan yöntem ve mimari farklıdır.

SQL Server da trigger'lar, T-SQL ile oluşturulabildiği gibi, Management Studio ile de oluşturulabilir. SQL Server, **DML (Data Manipulation Language)** komutları olan **INSERT**, **UPDATE**, **DELETE**'e destek verdiği gibi, **DDL (Data Definition Language)** komutları olan **CREATE**, **ALTER**, **DROP** sorguları için de trigger oluşturmayı destekler. DDL trigger'lar sadece transaction'dan sonra devreye girebilir, **INSTEAD OF** olamazlar.

## TRIGGER'LAR NASIL ÇALIŞIR?

Trigger'lar bir transaction olarak çalışırlar. Hata ile karşılaşıldığında **ROLLBACK** ile yapılan işlemler geri alınabilir. Trigger yapısının en önemli unsurlarından biri sözde tablolardır. Sözde tablolar, **INSERTED** ve **DELETED** olmak üzere iki adettir. Bu tablolar RAM üzerinde mantıksal olarak bulunurlar. Gerçek veri tablosuna veri eklendiğinde, eklenen kayıt **INSERTED** tablosuna da eklenir. Tablodan bir kayıt silindiğinde ise silinen kayıt **DELETED** tablosunda yer alır. Trigger'lar güncelleme işlemi için **UPDATED** isimli bir sözde tabloya sahip değildir. Güncellenen veriler ilk olarak silinerek **DELETE** işlemi ve sonrasında tekrar eklenerek **INSERT** işlemi gerçekleştirilir. Bir kayıt güncellendiğinde orijinal kayıt **DELETED** tablosunda, değişen kayıt ise **INSERTED** tablosunda saklanır.

### ON

Trigger'ın hangi nesne için oluşturulduğunu belirtmek için kullanılır.

### WITH ENCRYPTION

View ve Stored Procedure böümlerinde anlatılan özellikler aynen geçerlidir. Bu özellik sadece View ve Stored Procedure için geçerlidir. Hangi veriler üzerinde

trigger işlemi yapıldığının bilinmemesi gereken durumlarda, yani trigger'in yaptığı işlemlerin bilinmemesi için, içeriği şifreleme amacı ile kullanılır.

## WITH APPEND

SQL Server 6.5 ve öncesi versiyonlar için uyumluluk açısından desteklenmektedir. Bu eski versiyonlarda, aynı tablo üzerinde aynı işlemi gerçekleştiren birden fazla trigger oluşturulamaz. Bir tabloda veri ekleme ve güncelleme işlemi için oluşturulan bir trigger varsa, aynı tablo üzerinde bir başka güncelleme işlemi için yeni bir trigger oluşturulamaz.

Yeni versiyonlarda bu durum bir sorun değildir. Ancak geriye dönük çalıştırılması gereken veritabanlarında **WITH APPEND** özelliği, aynı tabloda aynı işlemi yapan birden fazla trigger oluşturulmasını sağlar.

## NOT FOR REPLICATION

Bu özellik ile replikasyon-iliskili görev tabloyu değiştirdiğinde, trigger başlatılmayacaktır.

## AS

Trigger'in içerik, yani script kısmına geçildiğini belirtir. Bu sözcükten sonraki sorgular trigger'in içeriğidir.

## FOR | AFTER İFADESİNE VE INSTEAD OF Koşulu

Trigger'i, **INSERT**, **UPDATE** ve **DELETE** gibi sorguların hangisi ya da hangileri tarafından ve ne zaman başlatılacağını belirtmek için kullanılır.

## TRIGGER TÜRLERİ VE INSERTED, DELETED TABLOLARI

Trigger'lar farklı ihtiyaçlara yönelik farklı türlere sahiptir. Bir trigger'in, veri ekleme, güncelleme ya da silme işleminden sonra tetiklenmesi gerektiği durumlarda farklı seçenekler tanımlanması gereklidir. Tanımlanan bu seçenekler ile trigger tam olarak ne yapacağını bilebilir.

Trigger'lar, veri ekleme, silme ve güncelleme işlemlerini gerçekleştirmek için **INSERTED** ve **DELETED** sözde tablolarına ihtiyaç duyur. İlerleyen anlatımlarda bu tablolara da değinerek çeşitli örnekler yapacağız.

## **INSERT TRIGGER**

**FOR INSERT** ile kullanılır. Tabloya yeni bir veri eklemek istendiğinde tetiklenen trigger türüdür. Eklenen her yeni satır, trigger var olduğu sürece var olan **INSERTED** tablosuna kaydedilir. **INSERTED** tablosu trigger'lar ile bütünlük bir kavramdır. Bir trigger'dan önce de yoktur, sonra da. Sadece trigger varken vardır.

## **DELETE TRIGGER**

**FOR DELETE** ile kullanılır. Tablodan bir veri silme işlemi gerçekleştirilirken kullanılır. Silinen her kaydın bir kopyası **DELETED** tablosunda tutulur. Trigger ile kullanılır.

## **UPDATE TRIGGER**

**FOR UPDATE** ile kullanılır. Tablodaki güncelleme işlemleri için tetikleme gerçekleştirmede kullanılır. Güncelleme işlemlerine özel bir **UPDATED** tablosu yoktur. Bunun yerine, güncellenen bir kayıt, tablodan silinmiş ve daha sonra tekrar eklendi olarak düşünülür. **INSERT** ve **DELETE** işlemlerini gerçekleştirdiği için doğal olarak **INSERTED** ve **DELETED** tablolarından erişilir. **INSERTED** ve **DELETED** tabloların satır sayıları eşittir.

## **TRIGGER OLUSTURMAK**

SQL Server'da iki farklı türde trigger oluşturulabilir. **FOR** ve **AFTER** ile transaction'dan sonra tetiklenecek, **INSTEAD OF** ile de tablolar değişiklikten etkilenmeden önce tetiklenecek trigger'lar oluşturulabilir. Bu DML trigger'ları **INSERT**, **UPDATE**, **DELETE** işlemleri için oluşturulabilir.

Trigger'lar tüm veritabanı nesneleri gibi **CREATE** ifadesiyle oluşturulur. **CREATE TRIGGER** ifadesini çalıştırabilmek için şu veritabanı izinlerinden birine sahip olunmalıdır.

- **sysadmin** server rolü
- **db\_owner** sistem rolü
- **db\_dlladmin** sistem rolü

Kullanıcının sahibi olduğu view ya da tablo üzerinde herhangi bir ek izne sahip olmasına gerek yoktur.

Trigger'lar otomatik tetiklenen bir nesne olması nedeniyle içerisinde kullanılabilecek ifadelerde bazı kısıtlamalar olması veri güvenliği ve tutarlılığı açısından gereklidir.

Trigger'lar şu ifadeleri içermemelidir;

- |                    |               |
|--------------------|---------------|
| • CREATE DATABASE  | • LOAD LOG    |
| • ALTER DATABASE   | • RESTORE LOG |
| • DROP DATABASE    | • DISK INIT   |
| • LOAD DATABASE    | • DISK RESIZE |
| • RESTORE DATABASE | • RECONFIGURE |

## **Söz Dizimi (DML Trigger)**

---

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
[WITH seçenekler]
{FOR | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
AS
-- Trigger SQL gövdesi
```

---

## **Söz Dizimi (DDL Trigger)**

---

```
CREATE TRIGGER trigger_ismi
ON { DATABASE | ALL SERVER }
{ veritabani_seviyeli_olaylar | server_seviyeli_olaylar }
AS
-- Trigger SQL gövdesi
```

---

## **INSERT TRIGGER**

Bir tabloya yeni kayıt ekledikten sonra devreye girecek işlemler için kullanılır. Trigger oluşturulduktan sonra, yeni eklenen her kaydı **Inserted** tablosunda tutmaya başlar. Bu kayıtlar gerçek tablonun yapısal bir kopyasıdır. SQL Server, satır bazlı trigger destek vermez. Her eklenen kayıt için değil, kayıt seti için çalışır.

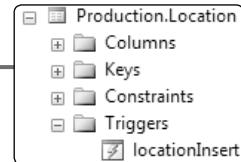
`Production.Location` tablosuna yeni bir lokasyon bilgisi eklemek istendiğinde, ekleme işleminden sonra çalışacak basit bir trigger oluşturalım.

---

```
CREATE TRIGGER locationInsert
ON Production.Location
AFTER INSERT
AS
BEGIN
SET NOCOUNT ON;
SELECT 'Yeni lokasyon bilgisi eklendi.';
SET NOCOUNT OFF;
END;
```

---

Oluşturulan trigger, **Management Studio** içerisinde, ilgili tablonun **Triggers** menüsünde görüntülenebilir.



Trigger'in tetiklenmesi için bir kayıt ekleyelim.

---

```
INSERT INTO Production.Location(Name, CostRate, Availability, ModifiedDate)
VALUES ('Yeni Lokasyon', 0.00, 0.00, GETDATE());
```

---

**INSERT** sorgusu ile birlikte kayıt eklenecektir. Ekleme işleminden hemen sonra da, ekranда aşağıdaki gibi bir bilgi mesajı görüntülenecektir.

(No column name)
1 Yeni lokasyon bilgisi eklendi.

Yeni eklenen her personel için bir bildirim zorunluluğu oluşturmak istenebilir. Bunu, trigger içerisinde **RAISERROR** kullanarak gerçekleştirebiliriz. Bu örnekte, bir personel yerine ürün ya da sipariş ekleme gibi bir senaryoda söz konusu olabilirdi.

---

```
CREATE TRIGGER trg_PersonelHatirlatici
ON Personeller
AFTER INSERT
AS
RAISERROR ('Eklenen Personeli Bildir', 16, 10);
```

---

Bu örneğimizde daha önceki örneklerde oluşturduğumuz `Personeller` tablosunu kullandık. Trigger ismi `trg_PersonelHatirlatici` olarak belirlendi.

```
Msg 50000, Level 16, State 10, Procedure trg_PersonelHatirlatici, Line 5
Eklenen Personeli Bildir
```

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

Yukarıdaki kullanımda hata seviyesi olarak 16 verildiği için kırmızı yazı ile hata bildirimleri yapılır. Ancak bu seviyeyi 10 yaparak bir uyarı bildirimini seviyesine düşürebiliriz.

```
Eklenen Personeli Bildir
```

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

Bu örnekteki işlem, her eklenen personelin mail ile yöneticiye bildirilmesi gibi bir işi de gerçekleştirmek istiyor olabilir.

Bir çok durumda, yapılan işlemlerin anlık olarak otomatik loglama ihtiyacı duyulur. Bu ihtiyacı karşılamak için trigger kullanılabilir.

**Production.Product** tablosunda yapılan her veri ekleme işlemini otomatik olarak **ProductLog** tablosuna kaydedelim.

**ProductLog** tablosunu oluşturalım.

---

```
CREATE TABLE ProductLog
(
    ProductID INT,
    Name VARCHAR(50),
    ProductNumber NVARCHAR(25),
    ListPrice DATETIME
);
```

---

**Production.Product** tablosunda anlık loglama içerisinde, bulunmasını istediğimiz sütunları barındıracak **ProductLog** tablosunu oluşturduk. Bu tabloda belirtilen sütunların birebir karşılığını, **INSERTED** isimli sözde tablodan alacağız.

---

```
CREATE TRIGGER trg_ProductLog
ON Production.Product
AFTER INSERT
AS
BEGIN
    INSERT INTO ProductLog SELECT ProductID, Name,
        ProductNumber, ListPrice
    FROM inserted;
END;
```

---

Oluşturulan trigger **Management Studio**'da, **Production.Product** tablosu içerisindeki **Triggers** bölümünde görülebilir.

Trigger'i test etmek için veri ekleyelim.




---

```
INSERT INTO Production.Product
    (Name, ProductNumber, MakeFlag, FinishedGoodsFlag,
    SafetyStockLevel, ReorderPoint, StandardCost, ListPrice,
    DaysToManufacture, SellStartDate, rowguid, ModifiedDate)
VALUES ('Test Ürün', 'SK-3335', 1, 0, 500, 700, 0, 0, 3,
    GETDATE(), NEWID(), GETDATE());
```

---

Normalde tek veri ekleme sorgusu çalışlığında tek bir “*1 row(s) effected*” mesajı döner. Ancak bu sorgu ile birlikte, iki kez aynı metnin **Messages** ekranında görüntülendiği görülür. Bunun nedeni; ilk metin **INSERT** sorgusunu belirtirken, ikinci sorgu trigger tarafından tetiklenerek, **ProductLog** tablosuna eklenen veri için otomatik olarak oluşturulan ve kullanılan **INSERT** sorgusunun mesajıdır.

---

```
SELECT ProductID, Name, ProductNumber FROM Production.Product
WHERE ProductNumber = 'SK-3335';
```

---

	ProductID	Name	ProductNumber
1	1021	Test Ürün1	SK-3335

---

```
SELECT * FROM ProductLog;
```

---

	ProductID	Name	ProductNumber	ListPrice
1	1021	Test Ürün1	SK-3335	1900-01-01 00:00:00.000

Trigger'i tetiklemek için sadece T-SQL ile değil, **Management Studio**'da kullanılabilir. Veri düzenleme ekranında yeni bir satır eklendiğinde trigger tetiklenecek ve yeni eklenen kaydı **ProductLog** tablosuna loglayacaktır.

Trigger'lar, veritabanı programlamaya yeni başlayanlara biraz karmaşık gelebilir. Bu nedenle anlamak ve ne tür çalışmalarında kullanılacağı kavramada zorlanılması normaldir. Trigger'lar, SQL Server'da ileri seviye bir kavramdır.

Farklı bir örnek daha yaparak **INSERTED** tablosunu ilişkisel olarak **JOIN** ile kullanalım.

`Production.Product` tablosuna son eklenen kaydı, `INSERTED` tablosundan okuyarak, otomatik olarak göstereceğiz.

---

```
CREATE TRIGGER trg_GetProduct
ON Production.Product
AFTER INSERT
AS
BEGIN
    SELECT PL.ProductID, PL.Name, PL.ProductNumber
    FROM Production.Product AS PL
    INNER JOIN INSERTED AS I
    ON PL.ProductID = I.ProductID;
END;
```

---

Trigger'ı tetiklemek için veri ekleyelim.

---

```
INSERT INTO Production.Product
    (Name, ProductNumber, MakeFlag, FinishedGoodsFlag,
    SafetyStockLevel, ReorderPoint, StandardCost, ListPrice, DaysToManufacture, SellStartDate,
    rowguid, ModifiedDate)
VALUES ('Test Product', 'SK-3334', 1, 0, 500, 700, 0.00, 0.00, 3,
    GETDATE(), NEWID(), GETDATE());
```

---

Veri eklendiğinde, son eklenen kayıt otomatik olarak listelenecektir.

ProductID	Name	ProductNumber
1	1033	Test Product SK-3334

## DELETE TRIGGER

Bir tablodan kayıt silindiğinde otomatik olarak gerçekleşmesi istenen işlemler için kullanılır. `DELETE` Trigger'ı tarafından `DELETED` isimli sözde tablo oluşturulur. `DELETED` tablosu sadece trigger içerisinde kullanılabilir. `DELETED` tablosu transaction log dosyasından türetilir. Bu nedenle silme teknigi önemlidir. `TRUNCATE` ile silme işlemini anlatırken, bu yöntemle silinen verilerin veri page'lerinden anında koparılarak silindiğini belirtmiştim. Yani, veri ile ilgili bir log tutulmaz. Bu nedenle `TRUNCATE` ile silinen veriler `DELETED` tablosu tarafından tutulmadığı için `TRUNCATE` ile silinen verilerde `DELETE` Trigger'ı beklenen işlemi gerçekleştiremez.

**DELETED** tablosu gerçek veri tablosundan bile silinmiş olan verileri tutar. Bunun nedeni, transaction sırasında geri alma işlemi gerçekleştirmeye olasılığında verilerin **ROLLBACK** ile geri alınabilmesini sağlamaktır. Hatırlarsanız, geri alma işlemleri için transaction log dosyası kullanılıyordu.

Kullanıcılar isimli tabloda silinen her verinin, **INSERTED** tablosundaki kayıtları kullanılarak bir mesaj gösterelim.

---

```
CREATE TRIGGER trg_KullaniciSil
ON Kullanicilar
AFTER DELETE
AS
BEGIN
    SELECT deleted.KullaniciAd + ' kullanıcı adına ve ' + deleted.Email +
        ' email adresine sahip kullanıcı silindi.' FROM deleted;
END;
```

---

**Kullanicilar** tablosundaki kayıt.

	KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com	02223456789

**Kullanicilar** tablosundan bir kayıt silelim.

---

```
DELETE FROM Kullanicilar WHERE KullaniciID = 1;
```

---

(No column name)
1 CihanOzhan kullanıcı adına ve cihan.ozhan@hotmail.com email adresine sahip kullanıcı silindi.

**DELETE** işleminden sonra verinin nasıl **SELECT** edilebildiği konusunu anlamak, **DELETE** trigger'larını anlamak için yeterlidir. Trigger içerisinde **Kullanicilar** tablosunu sorgulamak yerine, **DELETED** tablosunu sorguladık. Çünkü **Kullanicilar** tablosunda (gerçek veri tablosu) ilgili kayıt silindi. Ancak **DELETED** tablosu, silinen kayıtları transaction log dosyasını kullanarak tutar. Bu nedenle silinen kayda ulaşmak için **DELETED** tablosunu kullanabiliriz. Tabii ki **INSERTED** ile görüntülenen verinin gerçek tabloda aslinin olmadığını unutmamak gereklidir.

## UPDATE TRIGGER

Tablo üzerinde kayıtlarda güncelleme olduğunda tepki vermek üzere kodlanan trigger'lardır. **INSERT** ve **DELETE** trigger'lar gibi kendine has bir sözde tabloya

sahip değildir. **INSERTED** ve **DELETED** tablolarının her ikisini de kullanır. Ana tablodaki güncellenen kayıtların kopyasını **INSERTED** tablosu, güncellenmeden önceki hallerini ise **DELETED** tablosu tutar.

Gelişmiş veritabanı tasarımlarında, neredeyse tüm tablolar için güncellenme tarihi bilgisini tutacak bir sütun bulunur. Örneğin; bir ürün tablosunda, ürünün herhangi bir bilgisinin güncellenmesi durumunda, o anki sistem tarih ve zamanı ürünün güncellenme tarihi sütununa yazılır. Güncelleme işlemini yönetmek ve sorumluluk takibi açısından güncelleme yapan kullanıcının **ID** değeri de bir log tablosuna yazılır.

**Production.Product** tablosunda herhangi bir sütunun değeri değiştirildiğinde, **ModifiedDate** sütunu değerini o anki sistem tarih ve zaman bilgisile değiştirelim. Güncelleme işleminde, **ModifiedDate** sütunu **SET** edilmese bile, otomatik olarak o anki güncel bilgilerle değiştirilsin.

---

```
CREATE TRIGGER trg_UrunGuncellemeTarihiniGuncelle
ON Production.Product
AFTER UPDATE
AS
BEGIN
    UPDATE Production.Product
    SET ModifiedDate = GETDATE()
    WHERE ProductID = (SELECT ProductID FROM inserted)
END;
```

---

**ProductID** değeri 999 olan ürünü listeleyelim.

---

```
SELECT ProductID, Name, ProductNumber, ModifiedDate FROM Production.Product
WHERE ProductID = 999;
```

---

ProductID	Name	ProductNumber	ModifiedDate
1	999	Road-750 Red, 52	BK-R19B-52 2013-02-12 00:27:00.983

Şimdi, bu ürünü güncelleyelim.

---

```
UPDATE Production.Product
SET Name = 'Road-750 Red, 52'
WHERE ProductID = 999;
```

---

Güncellemeden sonraki sütun değerlerini listeleyelim.

```
SELECT ProductID, Name, ProductNumber, ModifiedDate FROM Production.Product
WHERE ProductID = 999;
```

	ProductID	Name	ProductNumber	ModifiedDate
1	999	Road-750 Yellow, 52	BK-R19B-52	2013-02-13 10:23:28.923

Güncelme işlemi sadece Name sütunu üzerinde yapılmasına rağmen ModifiedDate sütunu otomatik olarak trigger tarafından güncellendi.

Ürünler tablosundaki kayıtlarda bir güncelleme yapıldığında, bu güncelleme bilgilerini başka bir tabloda log olarak tutalım. Bu log tablosunda güncelleme işleminden önceki ve sonraki veriler tutulsun.

Güncellenen ürünlerin özet bilgilerinin loglanacağı tabloyu oluşturalım.

```
CREATE TABLE UrunGuncellemeLog
(
    ProductID INT,
    Name VARCHAR(50),
    ProductNumber NVARCHAR(25),
    ListPrice MONEY,
    ModifiedDate DATETIME
);
```

Otomatik loglama işlemini gerçekleştirecek **UPDATE** trigger'ı oluşturalım.

```
CREATE TRIGGER trg_UrunGuncelleLog
ON Production.Product
AFTER UPDATE
AS
BEGIN
    DECLARE @ProductID INT, @Name VARCHAR,
            @ProductNumber NVARCHAR, @ListPrice MONEY,
            @ModifiedDate DATETIME;

    SELECT @ProductID = i.ProductID, @Name = i.Name,
           @ProductNumber = i.ProductNumber, @ListPrice = i.ListPrice,
           @ModifiedDate = i.ModifiedDate FROM inserted AS i;
```

```
INSERT INTO UrunGuncellemeLog
VALUES (@ProductID, @Name, @ProductNumber, @ListPrice, @
ModifiedDate)
END;
```

---

Bir ürün güncelleyelim.

---

```
UPDATE Production.Product
SET Name = 'Road-750 Red, 52'
WHERE ProductID = 999;
```

---

Güncelleme işleminden sonra `UrunGuncellemeLog` tablosu aşağıdakine benzer değerlerde olacaktır.

	ProductID	Name	ProductNumber	ListPrice	ModifiedDate
1	999	R	B	539,99	2013-02-11 23:56:37.073
2	999	R	B	539,99	2013-02-12 00:27:00.983

Tek bir güncelleme yapılmasına rağmen `UrunGuncellemeLog` tablosuna iki kayıt eklendi. Bunun nedeni; `UPDATE` trigger'in `INSERTED` ve `DELETED` tablolarının her ikisini de kullanıyor olmasıdır. İlk sıradaki kayıt verinin güncellenmeden önceki halini (`DELETED`), ikinci kayıt ise, güncelleme sonrasında (`INSERTED`) yeni halini belirtir ve log olarak kaydeder.

## BİRDEN FAZLA İŞLEM İÇİN TRIGGER OLUŞTURMAK

Trigger'lar tekil işlemler için kullanılabildiği gibi, `INSERT`, `UPDATE` ve `DELETE` işlemlerinin tümünü tek bir trigger ile de takip edebilir.

Bunun diğer trigger'lardan tek farkı `AFTER` tanımlama kısmıdır.

---

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
AFTER INSERT, UPDATE, DELETE
```

---

`AFTER` ifadesinden sonra aralarına virgül konularak gerekli işlemlerin bildirimi yapılabilir.

## INSTEAD OF TRIGGER

**INSTEAD OF Trigger'lar** veri değişimi başlamadan hemen önce çalışırlar. Tabloda bir değişiklik yapılmadan, hatta constraint'ler bile devreye girmeden çalışırlar. Sözde tablolar (**INSERTED**, **DELETED**) bu trigger türü tarafından da desteklenir.

İşlem gerçekleşmeden hemen önce devreye girdiği için, bu trigger'lar genel olarak view'lere veri ekleme işlemleri yönetmek için kullanılır. Tablolar üzerinde de, yapılmak istenen işlemi durdurmak ya da farklı bir işleme yönlendirerek otomatik olarak belirlenen bu işlemin gerçekleşmesini sağlamak için kullanılabilir.

View bölümünde çoklu tablolar ile **JOIN** kullanılarak birleştirilen view'lere trigger'lar ile veri eklenebileceğini ancak zahmetli bir iş olduğunu belirtmişik. O konuda bahsedilen trigger türü **INSTEAD OF Trigger'**lardır.

Bir table üzerinde, her bir işlem türü (**INSERT**, **UPDATE**, **DELETE**) için sadece tek bir **INSTEAD OF Trigger** oluşturulabilir. Yani, bir tabloda **INSERT** işlemi için **INSTEAD OF Trigger** oluşturulduysa, aynı tabloda **INSERT** işlemi için ikinci bir **INSTEAD OF Trigger** oluşturulamaz.

**INSTEAD OF Trigger** örnekleri için ilgili tablo ve verileri oluşturalım.

**Müşteriler** tablosunu oluşturalım.

---

```
CREATE TABLE Musteriler
(
    MusteriID INT NOT NULL PRIMARY KEY,
    Ad      VARCHAR(40) NOT NULL
);
```

---

**Siparişler** tablosunu oluşturalım.

---

```
CREATE TABLE Siparisler
(
    SiparisID  INT IDENTITY NOT NULL PRIMARY KEY,
    MusteriID  INT NOT NULL REFERENCES Musteriler(MusteriID),
    SiparisTarih DATETIME NOT NULL
);
```

---

## Ürünler tablosunu oluşturalım.

---

```
CREATE TABLE Urunler
(
    UrunID INT IDENTITY NOT NULL PRIMARY KEY,
    Ad VARCHAR(50) NOT NULL,
    BirimFiyat MONEY NOT NULL
);
```

---

## SiparisUrunleri tablosunu oluşturalım.

---

```
CREATE TABLE SiparisUrunleri
(
    SiparisID INT NOT NULL REFERENCES Siparisler(SiparisID),
    UrunID INT NOT NULL REFERENCES Urunler(UrunID),
    BirimFiyat MONEY NOT NULL,
    Miktar INT NOT NULL
    CONSTRAINT PKSiparisUrun PRIMARY KEY CLUSTERED(SiparisID, UrunID)
);
```

---

## Tablolara örnek veri ekleyelim.

---

```
INSERT INTO Musteriler VALUES(1,'Bilişim Yayıncılığı');
INSERT INTO Musteriler VALUES(2,'Çocuk Kitapları Yayıncılığı');

INSERT INTO Siparisler VALUES(1, GETDATE());
INSERT INTO Siparisler VALUES(2, GETDATE());

INSERT INTO Urunler VALUES('İleri Seviye SQL Server T-SQL', 50);
INSERT INTO Urunler VALUES('Keloğlan Masalları', 20);

INSERT INTO SiparisUrunleri VALUES(1, 1, 50, 3);
INSERT INTO SiparisUrunleri VALUES(2, 2, 20, 2);
```

---

## Tablo ve verileri listeleyerek inceleyelim.

---

```
SELECT * FROM Siparisler;
```

---

	SiparisID	MusteriID	SiparisTarih
1	1	1	2013-02-12 11:37:10.397
2	2	2	2013-02-12 11:37:10.397

---

```
SELECT * FROM Urunler;
```

---

UrunID	Ad	BirimFiyat
1	İleri Seviye SQL Server T-SQL	50,00
2	Keloglan Masalları	20,00

---

```
SELECT * FROM Musteriler;
```

---

MusterID	Ad
1	Bilisim Yayıncılığı
2	Çocuk Kitapları Yayıncılığı

---

```
SELECT * FROM SiparisUrunleri;
```

---

SiparisID	UrunID	BirimFiyat	Miktar
1	1	50,00	3
2	2	20,00	2

Bu tablo ve verileri kullanarak **INNER JOIN** ile birleştirilen bir view oluşturacağız. Bu kitabın view bölümünde çoklu veri (**INNER JOIN**) için kullanılan view'leri anlatırken, view üzerinden veri ekleme işleminde bazı kısıtlamaları olduğunu belirtmiştık. Join ile birleştirilmiş bir view'in veri tablolarına **INSERT**, **UPDATE**, **DELETE** yapabilmek için **INSTEAD OF Trigger** kullanılır.

Yukarıda hazırladığımız örnek tablo ve verileri kullanarak bir view oluşturalım. **INSTEAD OF Trigger** örneğimizde bu oluşturulan view'i kullanarak **INSERT**, **UPDATE**, **DELETE** yapacağız.

---

```
CREATE VIEW vw_MusteriSiparisleri
AS
SELECT S.SiparisID,
       SU.UrunID,
       U.Ad,
       SU.BirimFiyat,
       SU.Miktar,
       S.SiparisTarih
  FROM Siparisler AS S
  JOIN SiparisUrunleri AS SU
```

```

ON S.SiparisID = SU.SiparisID
JOIN Urunler AS U
ON SU.UrunID = U.UrunID;

```

**vw\_MusteriSiparisleri** view'i, siparişlerin önemli bilgilerini listeleyecektir.

```
SELECT * FROM vw_MusteriSiparisleri;
```

SiparisID	UrunID	Ad	BirimFiyat	Miktar	SiparisTarih
1	1	İleri Seviye SQL Server T-SQL	50,00	3	2013-02-13 10:34:26.173
2	2	Keloglan Masalları	20,00	2	2013-02-13 10:34:26.173

## INSTEAD OF INSERT TRIGGER

Tablo ya da view'e fiziksəl olaraq veri ekleme işlemi yapılmadan önce veriyi inceleyerek, farklı bir işlem tercih edilmesi gereken durumlarda kullanılabilir.

**INSTEAD OF Trigger'lar** genel olarak view üzerinden veri ekleme işlemi için kullanılır. Bunun nedeni, view'in karmaşık sorgulara sahip olmasıdır. SQL Server, her ne kadar karmaşık sorguları çözümleyebilecek yeteneklere sahip olsa da, bu tür kullanıcı tercihi gereken ve veri bütünlüğü açısından risk oluşturabilecek durumlarda, kendisine yol göstermesi için T-SQL geliştiricisinin müdahale etmesini bekler.

**vw\_MusteriSiparisleri** view'ini kullanarak ilişkilendirilmiş tablolara veri eklemek için bir **INSTEAD OF INSERT** trigger oluşturmamızı.

```

CREATE TRIGGER trg_MusteriSiparisEkle
ON vw_MusteriSiparisleri
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON;
IF(SELECT COUNT(*) FROM inserted) > 0
BEGIN
INSERT INTO dbo.SiparisUrunleri
SELECT i.SiparisID,
i.UrunID,
i.BirimFiyat,
i.Miktar

```

```

    FROM inserted AS i
    JOIN Siparisler AS S
    ON i.SiparisID = S.SiparisID

    IF @@ROWCOUNT = 0
        RAISERROR('Eşleşme yok. Ekleme yapılamadı.', 10, 1)
    END;
    SET NOCOUNT OFF;
END;

```

---

Trigger'ın tetiklenebilmesi için bir veri ekleme gerçekleştireceğiz.

---

```

INSERT INTO vw_MusteriSiparisleri(
    SiparisID, SiparisTarih, UrunID, Miktar, BirimFiyat)
VALUES(1, '2013-02-02', 2, 10, 20);

```

---

View içeriğini listeleyelim.

SiparisID	UrunID	Ad	BirimFiyat	Miktar	SiparisTarih
1	1	İleri Seviye SQL Server T-SQL	50,00	3	2013-02-13 10:34:26.173
2	1	Keloglan Masalları	20,00	10	2013-02-13 10:34:26.173
3	2	Keloglan Masalları	20,00	2	2013-02-13 10:34:26.173

Veri ekleme işlemini bir tablo değil, view üzerinde gerçekleştirdik. Herhangi bir tetikleme ve ek işlem yapmamış olsak da, oluşturduğumuz trigger bu veri ekleme işleminin gerçekleşmesini sağladı.

Tüm çok tablolu view'ler için **INSTEAD OF Trigger** kullanılmak zorundadır demek yanlış olur. SQL Server'ın **Primary Key** sütunlar üzerinden sorunsuz olarak erişebileceğim view yapılarında trigger'lara gerek yoktur. Ancak gerek olmasa bile, veri bütünlüğü ve işin doğru yoldan yapılmaya gerekliliği açısından trigger kullanılması önerilir. **Join**'lı view'lerde genel olarak doğrudan veri ekleme işlemi hata ile sonuçlanır. Bu durumda **INSTEAD OF Trigger** kullanılır.

## INSTEAD OF UPDATE TRIGGER

**INSTEAD OF Trigger**'lar, güncelleme işleminden önce gerçekleştirilmesi gereken kontroller ve işlemlerdeki değişiklikler için kullanılabilir.

**Production.Product** tablosunda güncelleme işlemini yönetecek bir trigger oluşturalım.

---

```
CREATE TRIGGER trg_InsteadOfTrigger
ON Production.Product
INSTEAD OF UPDATE
AS
BEGIN
    PRINT 'Güncelleme işlemi gerçekleştirilmek istendi.';
END;
```

---

Bu trigger, güncelleme işleminden önce çalışacak ve güncellemenin gerçekleşmesini engelleyecektir.

`Production.Product` tablosunda `Name` sütununu değiştiren bir güncelleme işlemi yapalım.

İlk olarak güncellenecek kaydı görüntüleyelim.

---

```
SELECT ProductID, Name, ProductNumber FROM Production.Product
WHERE ProductID = 1;
```

---

ProductID	Name	ProductNumber
1	Adjustable Race	AR-5381

Kaydın `Name` sütununu değiştirelim.

---

```
UPDATE Production.Product
SET Name = 'Adjustable Race1'
WHERE ProductID = 1;
```

---

Güncelleme işlemi gerçekleştirilmek istendi.

(1 row(s) affected)

Güncelleme sorgusu hatasız çalışmasına rağmen kayıt güncellenmedi ve trigger içerisinde belirtilen uyarı mesajı görüntülendi.

## INSTEAD OF DELETE TRIGGER

Veri silme işlemlerinden önce de bazı işlem ya da kontroller yapılmak istenebilir. Örneğin; bir veri silme işlemi gerçekleştirilirken, silme işlemi yerine güncelleme işlemi yapmak gerekebilir.

Bu tür işlemleri gerçekleştirebilmek için **INSTEAD OF DELETE** trigger kullanılır.

**vw\_MusteriSiparisleri** view'ini kullanarak bir ürünün silinmek istediği senaryosunu ele alalım. Kullanıcı view üzerinden bir ürünü silmek istediginde, ürünün silinmesini değil, sadece yayından kaldırılmasını sağlayabiliriz. Genellikle veri bütünlüğünü sağlamak için birçok tabloda **AktifMi/isActive** gibi sütunlar bulundurulur. Bu sütunların veri tipi BIT olarak ayarlanır ve 0 değerini alırsa **False**, 1 değerini alırsa **True** olarak değerlendirilir. Yayında olması için bu değerin **True**, yani 1 olması beklenir.

Şimdi, **vw\_MusteriSiparisleri** view'ı üzerinden bir veri silme sırasında, veriyi silmek yerine ürünün **AktifMi** sütununu **False** yaparak yayından kaldırılmasını sağlayalım.

Senaryodaki işlemi gerçekleştirebilmek için ürünler tablosuna **AktifMi** sütunu ekleyelim.

---

```
ALTER TABLE Urunler
ADD AktifMi BIT;
```

---

Mevcut verilerdeki oluşan yeni sütunun değerini belirlemek için bir güncelleme yapalım.

---

```
UPDATE Urunler
SET AktifMi = 1;
```

---

Mevcut view içerisinde işlem yapacağımız için view'e **AktifMi** sütununu ekleyelim.

---

```
ALTER VIEW vw_MusteriSiparisleri
AS
SELECT S.SiparisID,
       SU.UrunID,
       U.Ad,
       SU.BirimFiyat,
       SU.Miktar,
       S.SiparisTarih,
       U.AktifMi
  FROM Siparisler AS S
 JOIN SiparisUrunleri AS SU
```

```

ON S.SiparisID = SU.SiparisID
JOIN Urunler AS U
ON SU.UrunID = U.UrunID;

```

---

View artık üzerinde veri silme işlemi gerçekleştirmeye hazır. View kullanarak veri silmeye çalışırken, silme işleminden önce devreye girecek ve **DELETE** yerine **UPDATE** sorgusu çalıştırarak ürünü silmeden, sadece **AktifMi** sütunu değerini False yapacak bir trigger oluşturalım.

---

```

CREATE TRIGGER trg_UrunuYayindanKaldir
ON vw_MusteriSiparisleri
INSTEAD OF DELETE
AS
BEGIN
SET NOCOUNT ON;
IF(SELECT COUNT(*) FROM deleted) > 0
BEGIN
DECLARE @ID INT
SELECT @ID = UrunID FROM deleted
UPDATE vw_MusteriSiparisleri
SET AktifMi = 0
WHERE UrunID = @ID;
IF @@ROWCOUNT = 0
RAISERROR('Eşleşme yok. Pasifleştirme işlemi yapılamadı.', 10, 1)
END;
SET NOCOUNT OFF;
END;

```

---

Trigger'ı tetikleyeceğim işlemi yapmadan önce, **Urunler** tablosundaki veriye bir göz atalım.

UrunID	Ad	BirimFiyat	AktifMi
1	İleri Seviye SQL Server T-SQL	50,00	1
2	Keloglan Masalları	20,00	1

View'i kullanarak ürün silelim.

---

```
DELETE FROM vw_MusteriSiparisleri WHERE UrunID = 1;
```

---

Silme işleminden sonra **Urunler** tablosundaki veriye tekrar göz atalım.

---

```
SELECT * FROM Urunler;
```

---

UrunID	Ad	BirimFiyat	AktifMi
1	İleri Seviye SQL Server T-SQL	50,00	0
2	Keloglan Masalları	20,00	1

**DELETE** sorgusuna rağmen verinin silinmediğini, sadece **AktifMi** sütununun değerinin değiştiğini görüyoruz.

Veri silme işlemlerinde doğrudan çok fazla kullanılmasına da, bu tür **DELETE** yerine farklı bir işlem yapmak için tercih edilen bir trigger türüdür. Birçok veritabanı için kritik ve arşiv niteliğinde kayıtlar vardır. Makale yayını yapan bir web sitesinin veritabanında, kullanıcının kendi yazdığı makaleleri yayından kaldırma hakkı olması doğaldır. Ancak silmek ile yayından kaldırma farklı işlemleridir. Veritabanından silinen bir kayıt üzerinde farklı işlemler ve nesneler tanımlanmış olabilir. Hatta istatistiksel anlamda bazı özel yazılımlar geliştirilmiş de olabilir. Verilerin tablolardan silinmesi, bu istatistiklerdeki verinin gerçek istatistikleri yansıtılmamasına sebep olur. Bu durumda, makaleyi tamamen silmek yerine yayından kaldırma daha doğru olacaktır. Yayından kaldırma işlemini tabi ki basit bir **UPDATE** ile de gerçekleştirebilirdik. Ancak **UPDATE** kullanmak, veritabanında ilgili tablo üzerine herhangi bir kullanıcı ya da geliştiricinin **DELETE** sorgusunu çalıştırmasını engellemez. Oluşturduğumuz bu trigger ile **DELETE** kullanılsa bile, verinin silinmediği garanti edilmiş olur.

**INSTEAD OF DELETE** trigger'ı için farklı bir örnek daha yapalım.



Bir tablo ya da view üzerinde aynı komutu (INSERT, UPDATE, DELETE) kullanan tek bir INSTEAD OF Trigger oluşturulabileceğini unutmayın. Yukarıdaki örneği yaptıysanız, şimdi yapacağımız örneği oluşturabilmek için DROP TRIGGER ile yukarıdaki trigger'ı silmeniz gereklidir.

---

Bir siparişi silerken, o sipariş ile ilişkili diğer kayıtları da silecek yeni bir trigger oluşturalım.

---

```
CREATE TRIGGER trg_MusteriSiparisSil
ON vw_MusteriSiparisleri
INSTEAD OF DELETE
AS
```

```

BEGIN
SET NOCOUNT ON;
IF(SELECT COUNT(*) FROM deleted) > 0
BEGIN
DELETE SU
FROM SiparisUrunleri AS SU
JOIN deleted AS d
ON d.SiparisID = SU.SiparisID
AND d.UrunID = SU.UrunID
DELETE S
FROM Siparisler AS S
JOIN deleted AS d
ON S.SiparisID = d.SiparisID
LEFT JOIN SiparisUrunleri AS SU
ON SU.SiparisID = d.SiparisID
AND SU.UrunID = d.SiparisID
END;
SET NOCOUNT OFF;
END;

```

---

Yukarıdaki trigger örneği ile view üzerinden bir sipariş silindiğinde, o siparişle ilişkili olan **SiparisUrunleri** listesindeki sipariş kaydı da silinecektir.

Trigger'ı tetiklemeden önce ilişkili tablolardaki kayıtları listeleyelim.

---

```

SELECT * FROM Siparisler;
GO
SELECT * FROM SiparisUrunleri;

```

---

	SiparisID	MusteriID	SiparisTarih	
1	1	1	2013-02-12 15:42:34.280	
2	2	2	2013-02-12 15:42:34.280	
	SiparisID	UrunID	BirimFiyat	Miktar
1	1	1	50,00	3
2	2	2	20,00	2



Şimdi yapacağımız silme işlemini gerçekleştirmeden önce, daha önce oluşturduğumuz **Urunler** ve **Siparisler** tablolarını **DROP** ile silerek tekrar oluşturun ve view'in hata üretmemesi için **Urunler** tablosuna **AktifMi** sütunu tekrar ekleyerek, daha önceki işlemleri gözden geçirin.

Trigger'ı tetikleyecek veri silme işlemini yapalım.

---

```
DELETE vw_MusteriSiparisleri WHERE SiparisID = 1 AND UrunID = 1;
```

---

Silme işleminden sonra, ilgili tablodaki verileri tekrar listeleyelim.

```
SELECT * FROM Siparisler;
GO
SELECT * FROM SiparisUrunleri;
```

---

	SiparisID	MusteriID	SiparisTarih	
1	2	2	2013-02-12 15:42:34.280	
	SiparisID	UrunID	BirimFiyat	Miktar
1	2	2	20,00	2

## IF UPDATE() VE COLUMNS\_UPDATED()

UPDATE trigger'da, bazen ilgilenilen sütunların zaten değişmiş olup olmadığıın bilinmesi gerekebilir. Belirlenen bu zaten değişmiş olan sütunlardan, trigger'in haberinin olması, trigger'da uygulanacak işlemlerin azaltılmasını sağlar. Yani trigger'a, kendi içerisinde bulunan sütunların güncellilik durumunu haber vererek trigger'in daha performanslı ve gerekli işlemleri uygulaması sağlanabilir.

## UPDATE() FONKSİYONU

UPDATE() fonksiyonu, trigger çalışma alanı içerisinde çalışan bir fonksiyondur. Belirli bir sütunun güncellenip güncellenmediğini öğrenmek için kullanılabilir. Güncelleme bilgisini Boolean veri tipiyle **True/False** olarak bildirir.

Örneğin; **Production.Product** içerisinde **ProductNumber** sütununun güncellenmeye karşı kontrol edilmesi gerekebilir. Bir ürünün ürün numarasının değişmesi veritabanı bütünlüğünü etkileyebilir. **ProductNumber** sütununun değiştirilmesini engellemek için trigger içerisinde **UPDATE()** fonksiyonunu kullanalım.

---

```
CREATE TRIGGER Production.ProductNumberControl
ON Production.Product
AFTER UPDATE
```

```

AS
BEGIN
IF UPDATE (ProductNumber)
BEGIN
PRINT 'ProductNumber değeri değiştirilemez.';
ROLLBACK
END;
END;

```

---

Yukarıdaki trigger oluşturulduktan sonra, **Management Studio** ya da **SQL** ile güncellenmeye karşı korumalı hale gelmiştir.

SQL ile güncellememeyi deneyelim.

---

```

UPDATE Production.Product
SET ProductNumber = 'SK-9283'
WHERE ProductID = 527;

```

---

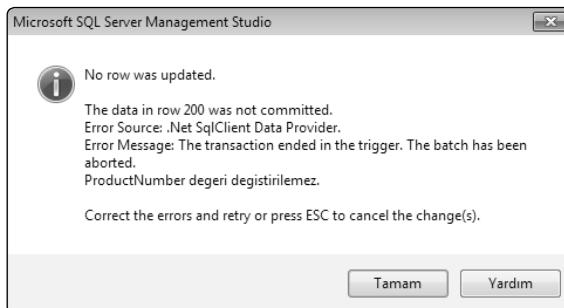
```

(1 row(s) affected)
ProductNumber degeri degistirilemez.
Msg 3609, Level 16, State 1, Line 2
The transaction ended in the trigger. The batch has been aborted.

```

Güncelleme işlemi başarısız oldu ve **PRINT** ile belirtilen mesaj ekranda görüntülendi.

**Management Studio** ile güncellememeyi deneyelim.



Her iki şekilde de artık bu sütun değeri değişikliklere karşı korumalıdır.

**Production.ProductNumberControl** trigger'i, **Production.Product** tablosunda yapılan güncelleme işlemlerini takibe eder. Güncelleme işlemleri içerisinde

`ProductNumber` sütunu varsa, bu sütunu korumaya alarak güncelleme işlemini reddeder.

Eğer ilgili tabloda farklı bir sütun üzerinde güncelleme gerçekleştirilirse, `UPDATE` sorgusu içerisinde `ProductNumber` sütunu olmadığı sürece güncelleme işlemi sorunsuz çalışacaktır.

## COLUMNS\_UPDATED() FONKSİYONU

Trigger ile güncelleme işlemlerinde kullanılır. `UPDATE()` fonksiyonu ile benzer amaca sahiptir. `UPDATE()` fonksiyonu tekil sütun değişikliklerini takip ederken, `COLUMNS_UPDATED()` fonksiyonu çoklu sütun değişikliklerini takip etmek için kullanılır. Bu fonksiyon, binary olarak değişen sütunların listesini metinsel veri tipi olan `VARCHAR` olarak döndürür.

### Söz Dizimi:

---

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
FOR UPDATE[, INSERT, DELETE]
AS
IF COLUMNS_UPDATED() & maskeleme_degeri > 0
BEGIN
    -- Sütun değişikliklerine göre çalışacak sorgu bloğu.
END;
```

---

## iç içe TRIGGER (NESTED TRIGGER)

Birbirini tetikleyerek çalışan trigger'lara iç içe trigger denir. Buradaki iç içe kavramı yanlış anlaşılmamalıdır. Bir trigger içerisinde başka bir trigger oluşturarak bunu tetiklemek anlamına gelmez. Zaten trigger'ların kullanıcı tarafından tetiklenmediğini, belirli olaylara karşı otomatik olarak tetiklendiğini biliyoruz.

Örneğin; bir işlem sonucunda tetiklenen trigger'in farklı tablo üzerinde gerçekleştirdiği veri ekleme ya da güncelleme, silme gibi herhangi bir işlemin sonucunda daha önceden tanımlanmış bir trigger'in tetiklenmesi durumuna iç içe trigger denir. Yani, bir trigger'in tetiklenmesiyle gerçekleşen işlemin üzerinde bulunduğu view ya da tablo'da, o trigger tarafından yapılan işlem için oluşturulmuş bir trigger'in otomatik olarak tetiklenmesidir.

Tetiklenen her trigger bir seviyedir. SQL Server, bu şekilde 32 seviyeye kadar iç içe trigger'ı destekler. Yani 32 trigger birbirini tetikleyebilir.

Trigger'ların seviyesini öğrenebilmek için `@@NESTLEVEL` ya da aşağıdaki söz dizimi kullanılabilir.

#### Söz Dizimi:

---

```
SELECT trigger_nestlevel(object_ID('trigger_ismi'));
```

---

İç içe trigger çok kullanılmayan ve açıkçası oldukça riskli bir kullanımındır. Trigger'lar otomatik tetiklenirler ve iç içe trigger'lardan herhangi bir trigger işleminde hata meydana gelirse, tüm trigger'ların yaptığı işlemler için `ROLLBACK` uygulanarak işlemler geri alınır.

SQL Server varsayılan olarak iç içe trigger'ı destekler. Ancak bu özelliğin kapatılması ya da tekrar açılması mümkündür.

Aşağıdaki yöntem ile iç içe trigger özelliği kapatılabilir.

---

```
sp_configure 'nested triggers', 0
```

---

İç içe trigger özelliğinin tekrar açılabilmesi için 0 değeri yerine 1 kullanmak yeterlidir.

## RECURSIVE TRIGGER

SQL Server'in varsayılan ayarlarında trigger'lar kendine atıflı olarak çalışmaz. Yani bir tablodaki herhangi bir sütun üzerinde yapılan değişiklik nedeniyle tetiklenen trigger, aynı tablo üzerinde başka bir sütunda değişikliğe neden oluyorsa, ikinci değişiklik için trigger tetiklenmez. Bu durumun tersine olarak tekrar tekrar çalışması yönetimi zorlaştırr.

Ancak, özel durumlar nedeniyle bazen esneklik gerekebilir. Bu özelliğinde tersi şekilde, tekrarlamaya izin vermek için aşağıdaki ayarlama yöntemi kullanılabilir.

---

```
ALTER DATABASE veritabani_ismi
SET RECURSIVE_TRIGGERS ON
```

---

## DDL TRIGGER'LAR

**DDL** (*Data Definition Language*), olarak tanınan **CREATE**, **ALTER**, **DROP** komutlarını içerir. Veritabanında nesne oluşturmak, değiştirmek ve silmek için kullanılan bu komutlar için de trigger'lar tanımlanabilir.

DDL trigger'lar, sadece **AFTER** trigger olarak tanımlanabilirler. **INSTEAD OF** türden bir DDL trigger oluşturulamaz.

Daha önce birçok örnek yaptığımız DML trigger'ları view ve tablolar üzerinde oluşturulduğu için, sadece veri değişimlerine karşı duyarlıdır. Amaçları, veri değişimindeki olayları yönetmektir. DDL trigger'lar ise veritabanı ve sunucu seviyeli olarak tanımlanabilirler. Örneğin; veritabanında bir tablo, prosedür ya da başka bir trigger'in oluşturulması gibi olaylarla ilgilenir ve bu olaylarla ilgili durumlarda tetiklenir.

DDL trigger'lar için de **ROLLBACK** ile işlemleri geri alma durumu söz konusu olabilir. DDL trigger'lar sadece  **eventdata()** fonksiyonu ile ortam hakkında bilgi alabilirler. Trigger içerisinde  **eventdata()** fonksiyonu çağrılarak, trigger'i tetikleyen olay ile ilgili ayrıntılı bilgiler içeren XML veri elde edilebilir.

### DDL Trigger Söz Dizimi:

---

```
CREATE TRIGGER trigger_ismi
ON {DATABASE | ALL SERVER}
FOR { veritabani_seviyeli_olaylar | sunucu_seviyeli_olaylar }
AS
-- Trigger sorgu gövdesi
```

---

## VERİTABANI SEVİYELİ DDL TRIGGER'LAR

Veritabanı seviyeli bir DDL trigger oluşturmak için **ON** komutundan sonra **DATABASE** kullanılmalıdır. DDL trigger'i oluştururken bir veritabanı ismi verilmez. Hangi veritabanı için çalıştırılırsa, o veritabanı için geçerli bir trigger olacaktır.

**AdventureWorks** veritabanı seçili iken yeni bir DDL trigger oluşturalım. Bu trigger, veritabanında tablo, prosedür, view oluşturmayı yasaklasın.

---

```
CREATE TRIGGER KritikNesnelerGuvenligi
ON DATABASE
FOR CREATE_TABLE, CREATE PROCEDURE, CREATE VIEW
AS
    PRINT 'Bu veritabanında tablo, prosedür ve view oluşturmak yasaktır!'
    ROLLBACK;
```

---

Oluşturulan bu trigger ile, artık **AdventureWorks** veritabanı üzerinde tablo, prosedür ya da view oluşturulamayacaktır.

Trigger'ı test etmek için bir tablo oluşturmayı deneyelim.

---

```
CREATE TABLE test ( ID INT );
```

---

```
Bu veritabanında tablo, prosedür ve view olusturmak yasaktır!
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

Tablo oluşturulmadı ve 16. seviyeden kritik bir hata olarak kırmızı hata mesajı fırlatıldı.

Veritabanı seviyeli DDL trigger'lar nesneler ile gerçekleştirilen işlemler için log tutma amacı ile de kullanılabilir.

**EventsLog** adında bir tablo oluşturalım.

---

```
CREATE TABLE EventsLog
(
    EventType VARCHAR(50),
    ObjectName VARCHAR(256),
    ObjectType VARCHAR(25),
    SQLCommand VARCHAR(MAX),
    UserName VARCHAR(256)
);
```

---

Oluşturacağımız DDL trigger'i, **AdventureWorks** veritabanı üzerinde oluşturulan tablo, view, procedure ve trigger nesnelerinin bilgilerini **EventsLog** tablosuna log olarak kaydetsin.

---

```

CREATE TRIGGER EventLogCreateBackup
ON DATABASE
FOR CREATE_PROCEDURE, CREATE_TABLE, CREATE_VIEW, CREATE_TRIGGER
AS
BEGIN
SET NOCOUNT ON;

DECLARE @Data XML;
SET @Data = EVENTDATA();

INSERT INTO dbo.EventsLog(EventType, ObjectName, ObjectType,
    SQLCommand, UserName)
VALUES (@Data.value('(/EVENT_INSTANCE/EventType) [1]', 'VARCHAR(50)'),
    @Data.value('(/EVENT_INSTANCE/ObjectName) [1]',
    'VARCHAR(256)'), @Data.value('(/EVENT_INSTANCE/ObjectType) [1]',
    'VARCHAR(25)'), @Data.value('(/EVENT_INSTANCE/TSQLCommand) [1]',
    'VARCHAR(MAX)'), @Data.value('(/EVENT_INSTANCE/LoginName) [1]', 'VARCHAR(256)')
);
SET NOCOUNT OFF;
END;

```

---

**EventsLog** tablosu ve **EventLogCreateBackup** isimli trigger'i test edebilmek için, önceki örnekte tablo ve view oluşturmayı engelleyen trigger'in kaldırılması gereklidir.

Trigger, **Management Studio** ile yandaki gibi kaldırılabilir. Sağda görünen trigger ismine sağ tıklayıp **Delete** butonuna tıklandığında trigger silinecektir.

Yeni bir tablo ve view oluşturalım.

---

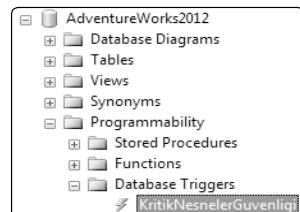
```

CREATE TABLE test1
(ID INT);

CREATE VIEW view1
AS
SELECT * FROM test1;

```

---



**EventsLog** tablosunu kontrol edelim.

---

```
SELECT * FROM EventsLog;
```

---

EventType	ObjectName	ObjectType	SQLCommand	UserName
1 CREATE_TABLE	test1	TABLE	CREATE TABLE test1 (ID INT);	dijibil-pc\dijibil
2 CREATE_VIEW	view1	VIEW	CREATE VIEW view1 AS SELECT * FROM test1	dijibil-pc\dijibil

Log tutulması istenen nesneler türünden nesne oluşumlarında, **EventsLog** tablosu log tutmaya devam edecektir. Bu işlem genel olarak **SQL Server DBA**'lerin veritabanının yönetimi ve güvenlik ayarlarını gözden geçirmesi için kullanılır.

## SUNUCU SEVİYELİ DDL TRIGGER'LAR

Sunucu seviyeli DDL trigger'lar, sunucunun bütününe özgü işlemlere duyarlıdır. Sunucu seviyeli trigger oluşturmak için **ON** komutundan sonra **ALL SERVER** kullanılmalıdır.

Yeni bir veritabanı oluşturma işlemini yakalayacak trigger oluşturalım.

---

```
CREATE TRIGGER SunucuBazlıDegisiklikler
ON ALL SERVER
FOR CREATE_DATABASE
AS
PRINT 'Veritabanı oluşturuldu。';
SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]', 'NVARCHAR(MAX)');
```

---

(No column name)	Veritabanı oluşturuldu.
1 CREATE DATABASE dbtest	(1 row(s) affected)

Benzer şekilde, yeni bir login oluşturma işlemi de yakalanabilir. Yukarıdaki trigger kullanımının **FOR** kısmını aşağıdaki gibi değiştirerek login oluşturma işlemleri yakalanabilir.

---

```
...
FOR DDL_LOGIN_EVENTS
...
```

---

Yeni bir login oluşturma işlemiyse aşağıdaki gibi yapılabilir.

---

```
CREATE LOGIN yakalanacak_login WITH PASSWORD = 'pwd123';
```

---

Sunucu bazlı trigger'lar `master.sys.server_triggers` sistem katalogunda yer alır. Oluşturulan trigger'ı listeleyelim.

---

```
SELECT * FROM master.sys.server_triggers;
```

---

	name	object_id	parent_class	parent_class_desc	parent_id	type	type_desc	create_date	modify_date	is_ms_shipped	is_disabled
1	SunucuBazliDegisiklikler	279672044	100	SERVER	0	TR	SQL_TRIGGER	2013-02-12 23:03:56.277	2013-02-12 23:03:56.277	0	0

## TRIGGER YÖNETİMİ

Trigger nesneleri, diğer tüm nesneler gibi değiştirilebilir, silinebilir. Hatta trigger'ların, aktiflik ve pasifliği değiştirilerek kullanımından kaldırılabilir ve tekrar kullanıma sunulabilir.

### TRIGGER'I DEĞİŞİSTİRMEK

Trigger'ların içeriği ve etkilendiği olaylar değiştirilebilir. `UPDATE` için hazırlanan bir trigger, `INSERT` ve `DELETE` işlemlerinde de tetiklenmesi için düzenlenenebilir. `sp_rename` sistem prosedürü kullanılarak trigger'ın sadece ismi değiştirilebilir.

`SunucuBazliDegisiklikler` trigger'ını değiştirek login oluşturma olaylarını da takip edecek şekilde düzenleyelim.

---

```
ALTER TRIGGER SunucuBazliDegisiklikler
ON ALL SERVER
FOR CREATE_DATABASE, DDL_LOGIN_EVENTS
AS
PRINT 'Veritabanı ya da Login oluşturma olayı yakalandı.';
SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
'NVARCHAR(MAX)');
```

---

Artık trigger'ımız login oluşturma işlemlerini de tetikleyebilir.

---

```
CREATE LOGIN testLogin WITH PASSWORD = '123';
```

---

(No column name)
1 CREATE LOGIN testLogin WITH PASSWORD = *****

## TRIGGER'LARI KAPATMAK VE AÇMAK

Trigger'ların tetiklenmesi geçici ya da kalıcı olarak engellenebilir. Bu işlem iki şekilde gerçekleştirilebilir. Bir trigger'in adını belirterek, tablo ya da view üzerinde sadece bu trigger'in tetiklenmeye kapatılması sağlanabileceği gibi, doğrudan tablo ya da view adı belirterek, bir trigger ismi belirtmeden tablo ya da view'deki tüm trigger'ların kapatılması sağlanabilir.

Aynı yöntemler ile kapatılan trigger ya da trigger'lar açılabilir.

Bir tablodaki tek bir trigger'i kapatalım.

---

```
ALTER TABLE Production.Product
DISABLE TRIGGER ProductNumberControl;
```

---

Bir tablodaki tüm trigger'ları kapatalım.

---

```
ALTER TABLE Production.Product
DISABLE TRIGGER ALL;
```

---

Kapatılan tek bir trigger'i tekrar açalım.

---

```
ALTER TABLE Production.Product
ENABLE TRIGGER ProductNumberControl;
```

---

Bir tablodaki tüm trigger'ları tekrar açalım.

---

```
ALTER TABLE Production.Product
DISABLE TRIGGER ALL;
```

---

## TRIGGER'LARI SİLMEK

Tablo ya da view üzerindeki trigger'ları silmek için tablo ya da view'in sahibi olmak ya da **sysadmin**, **db\_owner** rollerinden birine sahip olmak gereklidir.

---

```
DROP TRIGGER trigger_ismi
```

---

Bir tablo ya da view üzerinde tanımlı DML trigger'i, üzerinde bulunduğu tablo ya da view silindiğinde otomatik olarak silinir.

## VERİTABANI SEVİYELİ DDL TRIGGER'LARI SİLMEK

DML trigger'lardan farklı olarak, bir DDL trigger silmek için söz diziminin sonu **ON DATABASE** ile bitmek zorundadır.

### Söz Dizimi:

---

```
DROP TRIGGER trigger_ismi  
ON DATABASE
```

---

**EventLogCreateBackup** isimli veritabanı seviyeli DDL trigger'ı silelim.

---

```
DROP TRIGGER EventLogCreateBackup  
ON DATABASE;
```

---

## SUNUCU SEVİYELİ DDL TRIGGER'LARI SİLMEK

Sunucu seviyeli DDL trigger'ları silmek için söz dizimi **ON ALL SERVER** ifadesi alır.

### Söz Dizimi:

---

```
DROP TRIGGER trigger_ismi  
ON ALL SERVER;
```

---

**SunucuBazlıDegisiklikler** isimli sunucu bazlı DDL trigger'ı silelim.

---

```
DROP TRIGGER SunucuBazliDegisiklikler  
ON ALL SERVER;
```

---

# SORGU VE ERİŞİM GÜVENLİĞİ

SQL Server veritabanı yönetim sisteminde, veri ekleme, güncelleme, düzenleme, silme ve diğer yönetim fonksiyonlarını gerçekleştirecek ifadelerin tümü nesnedir. Nesnelerin erişimini yönetmek için izinler kullanılır.

Tüm işlemlerin nesne ve izinler ile yapılıyor olması, veritabanı programlama ve yönetimi daha disiplinli yapmayı zorunlu hale getirir. Ayrıca, veritabanında veri güvenliği, sadece nesne izinlerinin ayarlanması ile tamamlanmaz. Veriye izinsiz erişim ve geliştirici hatalarından dolayı gerçekleşen güvenlik zayıflıklarının de düzenlenmesi gereklidir.

**Sorgu ve Erişim Güvenliği** bölümünde ilk olarak, sorgu bazında veritabanı güvenliği kavramı ve SQL Injection gibi, önemli bir güvenlik zayıflığını inceleyeceğiz. Daha sonra, erişim güvenliği konusuna giriş yaparak, güvenlik ve erişim ile ilgili alınabilecek önlemleri inceleyeceğiz.

## SQL INJECTION

Veritabanı programcılarları veri ekleme, güncelleme, düzenleme ve silme gibi işlemler için istemci ve sunucu tarafında özel yazılımlar geliştirirler. Bir istemciden sunucuya ulaştırılan veri, istemci için hiç bir anlam ifade etmeyecek basit metinsel değerlerdir. Veritabanı programlama dilini tanıyan tek yapı, veritabanı motorudur. Bu nedenle, istemcide hatalı kod bile yazılsa, sorgu sunucuya gidene kadar hata alınmaz. Aynı zamanda, gönderilen SQL kodlarının, veritabanına ulaşana kadar, hangi işlemi yapabileceğini de bilinememesi anlamına gelir.

İstemci ve sunucunun, birbirine bu kadar yabancı olduğu mimaride, kötü niyetli kişiler, bu zafiyeti kullanarak, SQL kodları arasına kendi istedikleri ve gerçek sorgunun amacından farklı olarak çalışmasını sağlayacak komutlar enjekte (ekler) eder. Veritabanı özelliklerinin bu şekilde kötü niyetli kullanılmasına SQL Injection denir.

SQL Injection, veritabanı yönetim sisteminin mimarisinden çok, veritabanı programcısı ve yöneticisinin zafiyet ve bilgi eksikliğinden kaynaklanır. Veritabanı sunucu mimarinin derinliklerine inilirse, veritabanı yönetim sistemi yazılımının da önemli bir payı vardır. Ancak SQL Injection için kullanılan yöntemlerin neredeyse tamamı, tüm veritabanı yönetim sistemlerinde güvenli hale getirilebilir.

SQL Injection bölümünde, veritabanı programlama komutlarını kullanarak, izinsiz komut çalıştırma, sorgu arasına kod enjekte etme gibi yöntemleri inceleyeceğiz.

## SQL INJECTION KULLANIMI

SQL Injection, veritabanı sunucusuna gönderilen sorgu bloğuna, yeni bir sorgu sıkıştırma yöntemi ile gerçekleşir. Bu işlemi anlatabilmek için en temel haliyle bir SQL sorgusu oluşturalım.

Bir ürün sorgulamak için aşağıdaki sorgu kullanılır.

---

```
SELECT * FROM Production.Product WHERE ProductID = 1;
```

---

	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

Sorgu sonucunda, `ProductID` değeri 1 olan tek bir kayıt görüntülenir. Normal kullanımda bu sorguda bir hata ya da güvenlik zafiyeti yoktur. Ancak, veritabanı mimarisi ve motorunun çalışma modelini kötü niyetli olarak kullanmak mümkündür. Bu sorguya küçük bir ekleme yaparak tekrar çalışıralım.

---

```
SELECT * FROM Production.Product WHERE ProductID = 1 OR 1=1;
```

---

	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

Sorguya dışarıdan müdahale ettiğimiz için tüm kayıtlar listelendi. Senaryo bu kadar basit değildir. Veritabanı sorguları uygulama içerisinde çalıştırılmak üzere hazırlanırlar. Örneğin; bir web sitesinde kullanıcı bilgisi, QueryString aracılığı ile `ID` olarak yönetiliyor ise, aşağıdaki gibi bir URI oluşacaktır.

[www.dijibil.com/index.aspx?userID=1](http://www.dijibil.com/index.aspx?userID=1);

ASP.NET uygulamasında bu URI'nin anlamı, "1 UserID'li kullanıcıyı veritabanından getir" ifadesiyle eşdeğerdir.

ASP.NET tarafında bir önlem alınmadığı taktirde aşağıdaki gibi bir kullanım ile veritabanındaki tüm kullanıcılar web sitesinde listelenebilecektir.

[www.dijibil.com/index.aspx?userID=1' OR 1=1;--](http://www.dijibil.com/index.aspx?userID=1' OR 1=1;--)

Aynı şekilde, bu yöntem bir kullanıcı giriş formunda da kullanılabilir. Bu örnekleri incelemeye devam edeceğiz.

Daha önceki bölüm çalışmalarımızda `Kullanıcılar` adında bir tablo oluşturmuştuk. Bu tablonun yapısı ve içeriği aşağıdaki gibidir.

KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com 02223456789
2	2	testUser	sifre.test	user@test.com 2233567890

`Email` adresi üzerinden bir sorgu oluşturarak ilişkili kaydı getirelim.

---

```
SELECT * FROM Kullanıcılar WHERE Email = 'cihan.ozhan@hotmail.com';
```

---

KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com 02223456789

Sorgu sonucunda, tek bir kayıt getirilecektir. Çünkü kullanıcılar benzersiz email adresleri ile kaydedilirler.

Bu sorguya müdahale ederek tüm kullanıcıları getirebiliriz.

---

```
SELECT * FROM Kullanıcılar WHERE Email = 'dijibil' OR 'x'='x';
```

---

KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com 02223456789
2	2	testUser	sifre.test	user@test.com 2233567890

Tabloda iki kaydımız vardı. Bu kayıtların tamamı listelendi. Sorgu içerisinde yazdığımız `dijibii` metninin hiç bir özelliği yoktur. O kısım tamamen boş da bırakılabilirdi. Önemli olan; `OR` ve devamında eklediğimiz ifadedir.

Veritabanına boş değer göndererek tüm kayıtları listelemek, sizi şansa tabiidir miydi? O halde yapalım.

---

```
SELECT KullaniciAd, Email FROM Kullanicilar
WHERE KullaniciAd = '' OR ''='' AND Sifre = '' OR ''='';
```

---

KullaniciAd	Email
1 CihanOzhan	cihan.ozhan@hotmail.com
2 testUser	user@test.com

Sorgu içerisinde hiç bir değer göndermedik. Ancak tablodaki tüm kayıtları listeledik. SQL Injection’ı tehlikeli hale getiren özelliklerinden biriyse, neredeyse tüm sorguların çalıştırılabilir olmasıdır.

Basit ve masum bir `SELECT` cümlesi içerisinde `DROP TABLE` ifadesini ekleyerek, farklı bir tabloyu silmek yeteri kadar tehlikeli midir? Evet, bu daha başlangıç

---

```
SELECT * FROM Kullanicilar
WHERE Email = 'x'; DROP TABLE Urunler; --';
```

---

Bu sorgu çalıştığında `Urunler` tablosunun silindiğine emin olabilirsiniz.

SQL Injection, otomatik işlemler gerçekleştirebilecek yazılımlar ile **Brute Force** saldıruları da gerçekleştirmeye imkan verir. Gerekli güvenlik önlemleri alınmamış bir sorgu, otomatik bir algoritma ile deneme-yanılma yöntemiyle çalışacak bir saldırı silahı haline gelebilir.

---

```
SELECT * FROM Kullanicilar
WHERE Email = 'hacker@deneme.com' AND Sifre = 'merhaba123'
```

---

Bu sorgunun, programcı tarafından kullanılan sorgudan farkı yoktur. Ancak, dışarıdan gelen veri ve erişim sınırlanmadığı takdirde, kötü niyetli kullanıcı bu sorguyu istediği gibi çalıştırabilir. `Email` ve `Sifre` sütunlarına, otomatik olarak değişecek ve sürekli deneme yapacak bir yazılım ile sisteme erişim sağlamak isteyebilir. Doğru bilgiler bulunduğuanda, veritabanı `True` değerini döndürecektr.

SQL Injection ile bir **SELECT** sorgusu gerçekleştirken, sorgu arasında bir veri eklemesi gerçekleştirilebilir.

---

```
SELECT * FROM Kullanicilar
WHERE Email = 'x';
INSERT INTO Kullanicilar (KullaniciID, KullaniciAd, Sifre, Email, Telefon)
VALUES (5, 'saldırgan', 'sifre11', 'test@test.com', 01234567890);--';
```

---

KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com	02223456789
2	testUser	sifre.test	user@test.com	2233567890
3	saldırgan	sifre11	test@test.com	1234567890

Sorgu çalışır ve saldırıgın artık sistemin bir kullanıcı haline gelmiştir.

Saldırgan, benzer bir yapı kullanarak, eklediği kullanıcıyı güncelleyebilir.

---

```
SELECT * FROM Kullanicilar
WHERE Email = 'x';
UPDATE Kullanicilar
SET Email = 'hacked@test.com'
WHERE Email = 'test@test.com';
```

---

3	5	saldırgan	sifre11	hacked@test.com	1234567890
---	---	-----------	---------	-----------------	------------

Basit bir **KullanıcıAd** sorgulama SQL ifadesini, tehlikeli hale getirmek için **Escape** karakteri de kullanılabilir.

Normal sorgu aşağıdaki gibidir.

---

```
SELECT * FROM Kullanicilar
WHERE KullaniciAd = 'CihanOzhan';
```

---

SQL enjekte edilmiş hali aşağıdaki gibidir:

---

```
SELECT * FROM Kullanicilar
WHERE KullaniciAd = '\''; DROP TABLE Siparisler; --';
```

---

Escape ile sorgu kesildi ve hemen peşinde, **DROP TABLE** ifadesi çalıştırıldı.

Veritabanındaki izinler ve programsal yönetimin önemini anlamış olmalısınız.

Bu hataları yapmamak için neler yapılması gerektiğini ilerleyen kısımlarda inceleyeceğiz. Ancak, SQL Injection'ın neler yapabildiğini ve tehlikelerini biraz daha inceleyelim.

Geliştirici basit bir sorgu yazarak uzaktaki SQL Server sunucusu üzerinde bu sorguları çalıştırabiliyor. Bu tür canlı bir senaryoda yapılabilecek en önemli saldırılardan biri, veritabanı sunucusunu kapatmaktadır. Bir e-ticaret sitesinin veritabanı sunucusunu kapatma saldırısı yapıldığını düşünmek bile oluşabilecek felaketleri anlamaya yetecktir. Bu senaryo daha da geliştirilebilir. IIS'e kadar erişebilecek bir risk konusudur.

**SELECT** sorgusuna kod enjekte ederek veritabanı sunucusunu kapatalım.

---

```
SELECT KullaniciAd FROM Kullanicilar WHERE KullaniciAd = '';
SHUTDOWN WITH NOWAIT;
` AND Sifre='';
```

---

**SHUTDOWN** komutu, SQL Server'da yürürlükteki işlem ve oturumlar sonlandığında veritabanı sunucusunun kapatılmasını sağlar. Ancak, **SHUTDOWN WITH NOWAIT** olarak kullanılması, bekleme yapmadan hemen kapat anlamına gelir. Bu durumda, hiç bir işlem beklenmeden tüm işlemler kesilir ve sunucu kapatılır. Siparişler, ürün takipleri, ödemeler ve bu işlemleri gerçekleştirmek için oluşturulmuş transaction'ların sonucunda oluşabilecek sorunlar büyük olacaktır.



Dışarıdan aldığı bir saldırı ile, veritabanı sunucusu kapatılan bir e-ticaret firmasının iflas etmesi gerçekleşmiş bir olaydır. Bu tür saldırılardan etkilenilmesi, müşterilerin firmaya olan güvenini yitirmesini sağlar.

Durum bu kadarla da sınırlı değildir. SQL Server'da, veritabanı sunucusu içerisindeki işletim sistemini yönetmek için tasarlanmış nesneler ve ifadeler vardır. Bunların en önemlilerinden biri, **xp\_cmdshell**'dır. Varsayılan olarak, neredeyse hiç bir hosting firması bu nesneye erişim izni vermez. Ancak, kendi sunucunuza kullanıyorsanız güvenlik sizden sorulacaktır.

Basit bir SELECT sorgusuna **xp\_cmdshell**'i aktif edecek bir SQL enjekte edelim.

---

```
SELECT * FROM Kullanicilar
WHERE KullaniciAd = 'x' AND
      KullaniciID IS NULL;
EXEC sp_configure 'show advanced options',1;
RECONFIGURE;
EXEC sp_configure 'xp_cmdshell',1;
RECONFIGURE
```

---

Bu sorgu çalışırken **SQL Profiler** ile izlendiğinde, yaptığı işlemin aşağıdaki sorgu ile eş değer olduğu görülebilir.

---

```
EXEC sp_configure 'xp_cmdshell',1;
```

---

```
Configuration option 'xp_cmdshell' changed from 1 to 1. Run the RECONFIGURE statement to install.
```

**xp\_cmdshell** gibi, **xp\_regrepad** **Extented Sproc**'lar da benzer şekilde kullanılabilir. **xp\_regrepad** ile neler yapılabileceğine bakalım.

İşletim sisteminin versiyonunu öğrenelim.

---

```
DECLARE @IsletimSistemi VARCHAR(100)
EXECUTE xp_regrepad 'HKEY_LOCAL_MACHINE',
      'SOFTWARE\Microsoft\Windows NT\CurrentVersion', 'ProductName',
@IsletimSistemi OUTPUT;
PRINT @IsletimSistemi;
```

---

Windows 7 Ultimate

Sunucu adı, servis adı ve SQL Server'ın kullandığı port gibi birçok bilgi elde edilebilir.

---

```
SELECT @@SERVERNAME AS SunucuAd, @@SERVICENAME AS ServisAd;
DECLARE @value VARCHAR(20);
DECLARE @key VARCHAR(100);
IF ISNULL(CHARINDEX('\', @@SERVERNAME, 0), 0) > 0
BEGIN
  SET @key = 'SOFTWARE\Microsoft\Microsoft SQL Server\' + @@servicename
  + '\MSSQLServer\SuperSocketNetLib\Tcp';
END
```

```

ELSE
BEGIN
    SET @key = 'SOFTWARE\MICROSOFT\MSSQLSERVER\MSSQLSERVER\
SUPERSOCKETNETLIB\TCP';
END
SELECT @KEY as [Key]
EXEC master..xp_regrep
    @rootkey = 'HKEY_LOCAL_MACHINE',
    @key = @key,
    @value_name = 'TcpPort',
    @value = @value OUTPUT
SELECT 'Port Numarası : ' + CAST(@value AS VARCHAR(5)) AS
PortNumber;

```

---

	SunucuAd	ServisAd
1	DIJIBIL-PC	MSSQLSERVER
Key	1	SOFTWARE\MICROSOFT\MSSQLSERVER\MSSQLSERVER\SUPERSOCKETNETLIB\TCP
PortNumber	1	Port Numarası : 1433

Yukarıda incelediğimiz SQL Injection yöntemleri temel ve orta seviyedir. Sadece bu saldırısı yöntemleri ile geliştirici ve DBA'ların nelere dikkat etmesi gerektiğini inceledik. Daha ileri seviye ve karmaşık, uzun SQL enjeksiyon sorguları kullanılabilir.

## STORED PROCEDURE İLE SQL INJECTION

SQL Injection'ı engellemek için bilinen yanlışlardan biri de Stored Procedure kullanıldığı takdirde SQL Injection saldırılara maruz kalınmayacağıdır. Stored Procedure kullanımı bazı saldırıcıları yöntemlerini kısıtladığı için kod enjektesini de kısıtlayacaktır. Ancak önemli olan, prosedür içerisindeki kodların, enjeksiyona uğramayacak şekilde güvenli kod yazım teknikleriyle geliştirilmesidir.

Bir kullanıcı doğrulama prosedürü geliştirelim.

---

```

CREATE PROCEDURE KullaniciDogrula
    @kullaniciAd VARCHAR(50),
    @sifre        VARCHAR(50)

AS
BEGIN
    DECLARE @sql NVARCHAR(500);
    SET @sql = 'SELECT * FROM Kullanicilar
                WHERE KullaniciAd = ''' + @kullaniciAd + '''
                AND Sifre = ''' + @sifre + ''' ';
    EXEC(@sql);
END;

```

---

Prosedürü kullanarak bir kullanıcı girişini doğrulayalım.

---

```
EXEC KullaniciDogrula 'cihanozhan', 'cihan.sifre';
```

---

KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com 02223456789

Doğrulama işlemi başarıyla gerçekleşti. Ancak önceki tecrübelerimizden biliyoruz ki, sorgunun başarıyla çalışması enjeksiyona karşı da güvenli olduğu anlamına gelmez. Prosedür içerisinde kullanılan tek tırnaklı ve dışarıdan değişken ile alınan değeri doğrudan **SELECT** sorgusu içerisinde kullanma yöntemi bu prosedürü de enjeksiyona açık hale getirir.

Enjeksiyona karşı basit tedbirlerden biri olarak, tek tırnak kullanılmadan prosedür içeriği oluşturulmalıdır.

---

```

CREATE PROCEDURE KullaniciDogrula
    @kullaniciAd VARCHAR(50),
    @sifre        VARCHAR(50)

AS
BEGIN
    SELECT * FROM Kullanicilar
    WHERE KullaniciAd = @kullaniciAd
    AND Sifre = @sifre;
END;

```

---

Elbette, tek tırnak kullanımını engellemek tek başına yeterli değildir. Prosedür kullanımının gücü, prosedürün içerisinde, birçok programlama yöntemini kullanarak, dışarıdan alınan verinin işlenmesi ve belirli şartlara uygun hale getirilmesidir.

Prosedür ile enjeksiyon kullanımını incelemek için, daha önce oluşturduğumuz ve ürün arama işlemini gerçekleştiren `pr_UrunAra` prosedürünü ele alabiliriz.

---

```
ALTER PROCEDURE pr_UrunAra
    @ara VARCHAR(50)
AS
BEGIN
    DECLARE @sorgu VARCHAR(100)
    SET @sorgu = 'SELECT * FROM Production.Product
                  WHERE Name LIKE ''%' + @ara + '%''''
    EXEC (@sorgu)
END;
```

---



Daha önceki uygulamalarda oluşturduğumuz, `pr_UrunAra` prosedürünü kullanmadıysanız `ALTER` yerine `CREATE` kullanarak prosedürü oluşturabilirsiniz.

Prosedürün kullanımı aşağıdaki gibidir.

---

```
EXEC pr_UrunAra 'just';
```

---

	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

`SELECT` sorgusunu prosedür içeresine almamız SQL Injection işlemini engellemeyecektir. Bu prosedürün, aşağıdaki kod enjekte edilebilir `SELECT` sorgusundan bir farkı yoktur.

---

```
SELECT * FROM Production.Product WHERE Name LIKE '%just' or 1=1;--%;
```

---

Sorgu sonucunda, tüm ürünler listelendi.

Prosedürün, kod enjekte edilebilir olduğunu test edelim.

---

```
DECLARE @ara VARCHAR(20);
SET @ara = '%just' OR 1=1;--%';
EXEC pr_UrunAra @ara;
```

---

	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

just kriteri ile arama işleminde tek bir kayıt dönmesi gerekirken, tüm ürünler listelendi.

## SALDIRILARA KARŞI KORUNMA YÖNTEMLERİ

Sadece SQL saldırıları değil, yazılımlara birçok şekilde saldırı düzenlenebilmektedir. Bunların tespiti ve giderilmesi teknik olarak mümkündür. İyi bir yazılım ve veritabanı geliştiricisi için, gerekli tespitleri yapmak ve bu güvenlik açıklarını gidermek için bazı püf noktalarına degeneceğiz.

### KARAKTER FILTRELEYİN

Gerçekleştirdiğimiz SQL Injection test çalışmalarının birçoğunda özel karakterler kullandık. Bu karakterlerin çoğu, SQL Server tablosundaki veri içerisinde kullanılmayacaktır. Örneğin; bir isim ya da email bilgisi içerisinde tek tırnak ya da boşluk gibi karakterler kullanılmaz. İlgili sütunlara gönderilecek bu tür verilerin filtrelenmesi gereklidir. Genel olarak yazılımlarda özel karakterler filtrelenirler. İyi bir güvenlik için bu karakterlerin filtrelenmesi, SQL Injection saldırılarını önlemede önemli bir fayda sağlar.

Veri filtreleme işlemi, uygulama ve veritabanı olmak üzere iki katmanda da yapılabilir. Uygulama yazılımı katmanında, kullanıcıdan alınan hiç bir veri, kontrol edilmeden ve filtreleme işlemine tabi tutulmadan veritabanına gönderilmemelidir. Veritabanı katmanında da, uygulama katmanından gelen veriler hiç filtrelenmemiş gibi düşünülerek filtreleme işlemeye tabi tutulmalıdır. Filtreleme işlemi, iki katmanda da performans açısından küçük bir kayba neden olsa da, kazandırdığı veri güvenliğinin yanında bu performans kaybı önemli değildir.

Geniş çaplı uygulamalarda veritabanı ve uygulama katmanın geliştiricileri farklıdır. Bazen iki katmanda da birden fazla geliştirici görev alabilmektedir.

Filtreleme ve güvenlik görevini, veritabanında performans kaybı yaşamamak için uygulama yazılımı tarafına bırakmak doğru değildir. Çünkü her geliştiricinin güvenlik ve yazılım bilgisi, bu önlemleri almak için yeterli olmayabilir. Ya da bu önlemleri almaya gerek duymayabilir. Halbuki bu tür önlemler bir güvenlik standardı olarak kullanılmalıdır.

SQL Server, metinsel verilerin düzenlenmesi için birçok fonksiyona sahiptir. Filtreleme işlemlerinde bu fonksiyonlar kullanılarak,filtreleme işlemleri kolay ve hızlı hale getirilebilir.

Örneğin; kullanılan tek tırnakları çift tırnağa dönüştürmek için `REPLACE` fonksiyonu kullanılabilir.

#### **Orjinal Metin: 'SQL' Injection**

---

```
SELECT REPLACE('''SQL'' Injection','''','''');
```

---

#### **İşlenmiş Metin: "SQL" Injection**

Dikkat edilmesi gereken önemli bir konu da, sütun ya da benzeri nesne isimlerinin aralarında boşluk olmamasıdır. **Management Studio**'da bir sütun oluştururken isim olarak arasında boşluk olan bir isim belirlendiğinde, SQL Server otomatik olarak bu nesne ismini kare parantez ( [ ] ) içerisinde alacaktır. Bu kural, geliştiriciler için de geçerli olmalıdır.

Bir metni kare parantez içerisinde almak için `QUOTENAME` fonksiyonu kullanılabilir.

---

```
SELECT QUOTENAME('Kullanıcı Adı');
```

---

	(No column name)
1	[Kullanıcı Adı]

Kullandığımız bu fonksiyonlar gibi, metin içerisinde ve kenarlarından boşlukları silen fonksiyon gibi birçok farklı fonksiyon kullanılabilir.

## **KAYIT UZUNLUKLARINI SINIRLAYIN**

SQL saldırısında sorgu kayıt uzunluğu önemlidir. Tablodaki bir sütun için 10 karakterlik bir uzunluk ayrıldıysa, sorguların da bu karakter uzunluğundan fazla bir değeri kabul etmemesi gereklidir.

Karakter uzunluğu konusu mimari açıdan şu şekilde ele alınmalıdır. Veritabanı katmanı ve uygulama katmanı arasında, sorgu ve servis katmanları bulunabilir. En alt katman olan veritabanındaki sütun uzunluğu 10 karakter ile sınırlıysa, tüm sorgularda ve bu alan için oluşturulan değişkenlerde de 10 karakterlik bir uzunluk kullanılmalıdır. Aynı zamanda, oluşturulan servisler ve uygulama arayüzünde, kullanıcının veri girişi yaptığı ekranlardaki veri giriş kontrollerinde (Input, TextBox vb.) de 10 karakter uzunluk sınırı bulunmalıdır.

## **VERİ TIPLERİNİ KONTROL EDİN**

Kayıt uzunlığında olduğu gibi, uygulama katmanı, veritabanı katmanı ve ara katmanlardaki aynı işi yapan tüm veri tipleri aynı olmalıdır. Kullanıcıdan sayısal bir değer alınması gerekiyorsa, metinsel ya da bir tarih türünü değer olarak almamalıdır. Bu işlemin uygulama katmanındaki yazılım ile kontrol edilmesi gereklidir.

## **YETKİLERİ SINIRLANDIRIN**

Bölümün ilerleyen kısımlarında inceleyeceğimiz izinler ve yetkilerin minimum seviyede olması önerilir. Bir kullanıcı, sadece **SELECT** ile veri seçme işlemi gerçekleştirmeye yetkiye sahipse; **INSERT**, **UPDATE** ve **DELETE** gibi izinlere sahip olmaması gereklidir.

## **UYGULAMALARI TARAYIN**

Ne kadar dikkat edilirse edilsin, yazılımlarda mutlaka güvenlik açıkları ve yazılımsal hatalar olacaktır. Bu hataların giderilmesi için tespit edilmesi gereklidir. İş gücü, hız ve beceriklilik açısından bu işi belirli çerçevelerde insanlardan daha verimli yapabilecek açık tarama ve güvenlik test uygulamaları mevcuttur. Bu tür uygulamalar ile güvenlik analizleri yapılarak, zafiyet noktaları belirlenmeli ve geliştirme ekipleriyle ilgili güvenlik zafiyetleri giderilmelidir.

# **ERİŞİM GÜVENLİĞİ**

SQL Server 2012 veritabanı, geliştirme ortamı, kurum ve işletmeler tarafından kullanılmaktadır. Bu kuruluşların, bir çok işlem için kullanılan veritabanları, özel ve gizliliği yüksek, korunması gereken verilerin tutulduğu veritabanları vardır. Bir kuruluşun mali gelir-gider hesapları ya da en özel örneklerden biri olan, bankaların milyonlarca kullanıcısının hesap bilgileri, hesap hareketleri gibi çok özel bilgiler veritabanlarında tutulur.

Bu kadar özel bilgilerin tutulduğu veritabanlarının normal kullanıcılar ve kuruluş çalışanları için, çeşitli izin ve kısıtlamalar dahilinde kullanıma açılmalı ve dışarıdan izinsiz erişmeye çalışacak kötü niyetli kişiler (bilgisayar korsanı) için verilere erişilemez hale getirilmelidir.

Tüm bu süreci yönetmesi gereken kişi SQL Server DBA'dır. Veritabanı yöneticisi, bu izin ve yetkileri ayarlamak için kullanıcı oturumları oluşturur, oturum izinlerini yapılandırır ve roller atar. Atanan izin ve roller ile kullanıcıların hangi nesne ve veri kümelerine erişebileceğini, bu veriler üzerinde ne tür işlemler yapabileceği ayarlanmalıdır.

Güvenliği yönetirken hedef şunlar olmalıdır.

- Kullanıcıların veriye erişim ihtiyacı ile sizin veriye yetkisiz erişimi engellemeye ihtiyacını dengeleyin.
- Veritabanı izinlerini, kullanıcıların yetkisiz ve zararlı komutları çalıştırmasını zorlaştıracak hale getirin.
- Güvenlik denetimlerini sürekli hale getirin.
- Sadece veritabanı güvenliği yeterli değildir. Sistem ve network güvenliğinin yeterli olup olmadığı konusunda sistem ve network uzmanları ile koordineli ihtiyaç analizi ve güvenlik testleri gerçekleştirin.
- Kullanıcı, roller ve gruplardan kaynaklanabilecek, yanlış kullanıcıların yetkilendirilmesi gibi güvenlik zaafiyetlerini yönetin.

## **ERİŞİM GÜVENLİĞİNE GENEL BAKIŞ**

Birçok veritabanı nesne yönelimli olsa da temel nesne olarak farklı yapı ve yönetim modelini kullanır. SQL Server, veritabanındaki tüm nesneleri şemalarla yönetir. Her şema roller tarafından sahiplenir. Bu yapı, SQL Server içerisinde nesnel ilişkilendirmenin temellerini oluşturur. Bazı durumlarda şemalar ile tasarlanan sistemler içinden çıkışmaz alt sorunlara sebep olabilir. Bunun sebebi; sahiplik zincirinin hatalı oluşturulmasıdır.

## **İZİNLERİNİ ANLAMAK**

Veritabanı, güvenlik esaslarına verilebilecek izinlere sahiptir. Bu izinler, bir anahtar kelime ya da verilen izni belirten anahtar kelimelerle başlayabilir. Veritabanında veri erişimi ve yönetimini sağlamak için kullanılan izinler, nesnelerin erişim ve üzerinde işlem yapma yetenekleri kazanmasını sağlarlar.

## VERİLEN İZİNLERİ SINAMAK

Sunucu kapsamındaki her nesne belirli izinlere sahiptir. Bu izinlerin izin hiyerarşisi hakkında bilgi alabilmek için `sys.fn_builtin_permissions` isimli `built-in` fonksiyonu kullanılır.

### Söz Dizimi:

---

```
SELECT * FROM sys.fn_builtin_permissions( DEFAULT | NULL | nesne_ismi );
```

---

Yerleşik izinlerin tam listesini öğrenebilmek için, fonksiyona `DEFAULT` ya da `NULL` parametresi verilebilir.

---

```
SELECT * FROM sys.fn_builtin_permissions(default);
```

---

	class_desc	permission_name	type	covering_permission_name	parent_class_desc	parent_covering_permission_name
1	DATABASE	CREATE TABLE	CRTB	ALTER	SERVER	CONTROL SERVER
2	DATABASE	CREATE VIEW	CRVW	ALTER	SERVER	CONTROL SERVER
3	DATABASE	CREATE PROCEDURE	CRPR	ALTER	SERVER	CONTROL SERVER
4	DATABASE	CREATE FUNCTION	CRFN	ALTER	SERVER	CONTROL SERVER
5	DATABASE	CREATE RULE	CRRU	ALTER	SERVER	CONTROL SERVER
6	DATABASE	CREATE DEFAULT	CRDF	ALTER	SERVER	CONTROL SERVER
7	DATABASE	BACKUP DATABASE	BADB	CONTROL	SERVER	CONTROL SERVER

`default` parametresi yerine, `sys.fn_builtin_permissions(NULL)` olarak da kullanılabilirildi.

Bu fonksiyon ile bir nesnenin izin hiyerarşisini elde etmek için parametre olarak nesne ismi verilebilir.

`DATABASE` nesnesinin izinlerini görüntüleyelim.

---

```
SELECT * FROM sys.fn_builtin_permissions('DATABASE');
```

---

	class_desc	permission_name	type	covering_permission_name	parent_class_desc	parent_covering_permission_name
1	DATABASE	CREATE TABLE	CRTB	ALTER	SERVER	CONTROL SERVER
2	DATABASE	CREATE VIEW	CRVW	ALTER	SERVER	CONTROL SERVER
3	DATABASE	CREATE PROCEDURE	CRPR	ALTER	SERVER	CONTROL SERVER
4	DATABASE	CREATE FUNCTION	CRFN	ALTER	SERVER	CONTROL SERVER
5	DATABASE	CREATE RULE	CRRU	ALTER	SERVER	CONTROL SERVER
6	DATABASE	CREATE DEFAULT	CRDF	ALTER	SERVER	CONTROL SERVER
7	DATABASE	BACKUP DATABASE	BADB	CONTROL	SERVER	CONTROL SERVER

Bir login ya da bir sertifikanın da izinleri öğrenilebilmektedir.

```
SELECT * FROM sys.fn_builtin_permissions('CERTIFICATE');
```

	class_desc	permission_name	type	covering_permission_name	parent_class_desc	parent_covering_permission_name
1	CERTIFICATE	VIEW DEFINITION	VW	CONTROL	DATABASE	VIEW DEFINITION
2	CERTIFICATE	REFERENCES	RF	CONTROL	DATABASE	REFERENCES
3	CERTIFICATE	ALTER	AL	CONTROL	DATABASE	ALTER ANY CERTIFICATE
4	CERTIFICATE	TAKE OWNERSHIP	TO	CONTROL	DATABASE	CONTROL
5	CERTIFICATE	CONTROL	CL		DATABASE	CONTROL

İzinleri öğrenilebilecek nesnelerin bir listesini almak için aşağıdaki soru kullanılabilir.

```
SELECT * FROM sys.fn_builtin_permissions(default)
WHERE permission_name = 'SELECT';
```

	class_desc	permission_name	type	covering_permission_name	parent_class_desc	parent_covering_permission_name
1	DATABASE	SELECT	SL	CONTROL	SERVER	CONTROL SERVER
2	OBJECT	SELECT	SL	RECEIVE	SCHEMA	SELECT
3	SCHEMA	SELECT	SL	CONTROL	DATABASE	SELECT

## SQL SERVER KİMLİK DOĞRULAMA YÖNTEMLERİ

SQL Server, dışarıdan erişimleri kimlik doğrulama ile gerçekleştirir. SQL Server güvenlik modeli, iki tür kimlik doğrulama moduna sahiptir.

- **Windows Kimlik Doğrulaması (Windows Authentication Only):** Veritabanına, lokal ağ üzerinden erişebilmek için kullanılabilir.
- **Karışık Güvenlik (Mixed Security):** Yerel ağ dışından erişimlerde, uzak bağlantı ve Windows kullanılmayan alanlardan veritabanına erişim için kullanılabilir.

Bu güvenlik modları, sunucu seviyeli yapılandırma için kullanılır. Bu işlem sistem bazında değil, veritabanı sunucu kopyası bazında gerçekleştirilir. Yani, her sunucu kopyası için farklı güvenlik yapılandırılması gerçekleştirilebilir.

## WINDOWS KİMLİK DOĞRULAMASI

Windows kimlik doğrulama modu kullanıldığından, kimlik doğrulaması Windows etki alanındaki kullanıcı ve grup hesapları ile yapılır. Bu doğrulama yöntemi ile, kullanıcı oturum kimliği ve şifresi olmadan veritabanına erişilmesini sağlar. Bu durum, etki alanı kullanıcılar için bazı ek zahmetlerden kurtarması

nedeniyle yararlıdır. Kullanıcılar, Windows etki alanı ile bağlandıklarından dolayı, veritabanı erişimi için ek oturum ve şifre hatırlamak ya da bunları yönetmek zorunda değildir. Kullanıcı, bulunduğu Windows etki alanı güvenlik ayarları doğrultusunda izinlere sahiptir. Bu kullanım, SQL Server yönetici için de tercih edilebilecek ve iş yükünü azaltacak doğrulama yöntemidir. Çünkü veritabanı yönetici, her kullanıcı için ayrı oturum ve şifre oluşturarak bunlara rol ve izin atamaları yapmak zorunda kalmayacaktır. Bu yöntem ile herkes, hazırlanan çeşitli Windows grupları ile yönetilebilir. Windows kimlik doğrulama kullanıldığında, SQL Server kullanıcının hesap adı ya da grup üyeliğine göre otomatik olarak kimlik doğrulamasını gerçekleştirir. Kullanıcı ya da kullanıcının grubuna bir veritabanı için erişim hakkı verilirse, bu veritabanına erişim otomatik olarak o kullanıcıya verilir.

## KARIŞIK GÜVENLİK VE SQL SERVER OTURUMLARI

Karişik güvenlik (*Mixed Security*) ile hem Windows hem de SQL Server oturumları kullanılabilir. Genel olarak yerel ağın dışında kalan veritabanına bağlantı gereksinimleri bu yöntem ile giderilir. Örneğin; veritabanına internetten erişmek için kullanılır. Internetten erişecek uygulamaları otomatik olarak belirli bir hesap kullanması ya da kullanıcı adı ve şifre ile yetkilendirme yapılarak veritabanına erişim sağlanabilir.

## ÖZEL AMAÇLI OTURUMLAR VE KULLANICILAR

SQL Server erişimi sunucu oturumları ile yönetilir. Bu oturumlar, şu düzeylerde yapılandırılabilir.

- Oturumların ait olduğu rol ile
- Veritabanlarına erişim izni vererek
- Nesnelere erişim izni vererek ya da erişimi engelleyerek

Veritabanı erişimleri için farklı yetkilere sahip erişim grupları vardır. Bu gruplar, kendilerine dahil edilen kullanıcılarla yetkilerini kullanma hakkı atarlar. Ayrıca, **dbo**, **guest** ve **sys** kullanıcıları gibi bazı varsayılan kullanıcılar sayesinde, sunucudaki varsayılan ayarlar ile oluşturulmuş kullanıcılar da veritabanı yönetiminde kullanılabilir.

## **ADMINISTRATORS GRUBU İLE ÇALIŞMAK**

**Administrators** grubu, veritabanı sunucusu üzerinde yerel bir gruptur. SQL Server'da bu grup üyelerine, varsayılan olarak **sysadmin** sunucu rolü verilir. Bu grup üyeleri, yerel Administrator kullanıcı hesabını ve sistemi yerel olarak yönetecek şekilde ayarlanan kullanıcıları içerir.

## **ADMINISTRATOR KULLANICI HESABI İLE ÇALIŞMAK**

Sunucu üzerinde yönetici hakları sağlayan bir kullanıcı hesabıdır. SQL Server'da bu hesap, varsayılan olarak sysadmin sunucu rolünün üyesidir.

## **SA OTURUMU İLE ÇALIŞMAK**

SQL Server için, sistem yönetici hesabıdır. Bu oturum, varsayılan olarak **sysadmin** sunucu rolüne sahiptir. SQL Server kurulumunda varsayılan olarak gelir. **sa** artık gerekli bir hesap olmamakla birlikte, varsayılan olarak kalması isteniyorsa, karmaşık ve güçlü bir şifre atanmalıdır. SQL Server ile ilgilenen herkes iyi niyetli değildir. Bazı kötü niyetli uzmanlar da sa oturumunun farkındadır. Bu nedenle, açık hedef halindeki bu oturumu kullanmamanız, hatta iptal etmeniz ya da silmeniz önerilir. sa oturumu yerine, sistem yöneticilerini sysadmin sunucusu üyesi yapın. Bu şekilde, sistem yöneticileri veritabanı üzerinde istenilen ayarları yapabilir. Sunucuya erişim konusunda sorun yaşanırsa, yerel olarak sunucuya bağlanılıp, gerekli ayarlar yeniden yapılabılır.

## **NETWORK SERVICE VE**

## **SYSTEM OTURUMLARI İLE ÇALIŞMAK**

Sunucu üzerinde, yerleşik yerel hesaplardır. Sunucu hesabının kurulum ayarlarına bağlı olarak kullanılabilir durumdadırlar.

Örneğin; bir rapor sunucusu olarak yapılandırılan bir sunucunun **NETWORK SERVICE** hesabı için bir oturumu olur. Bu oturum; **master**, **msdb**, **ReportServer**, **ReportServerTempDB** veritabanları üzerinde özel veritabanı rolü olan **RSEexecRole** rolüne sahip olur.

## **GUEST KULLANICISI**

SQL Server'da bir oturumu bulunan herkesin, herhangi bir veritabanına erişebilmesi için veritabanına eklenen özel bir kullanıcıdır. Guest kullanıcı, herhangi bir veritabanına erişim hakkı bulunmayan, ancak SQL Server'a erişim

hakki bulunan kullanıcılar için tasarlanmıştır. Bu hesabın yetkileri sınırlıdır. Veritabanı oluştururken model olarak kullanılan model veritabanında Guest hesabı bulunması nedeniyle, yeni oluşturulan tüm veritabanlarında Guest kullanıcısı vardır. Bu kullanıcının, veritabanında bulunması zorunlu değildir ve veritabanından kaldırılabilir. Genel olarak kullanıcılar `master` ve `tempdb` veritabanlarına Guest kullanıcısı ile erişirler. Bu kullanıcının izin ve hakları kısıtlı olduğu için geniş yetki gereken işlemler için kullanılamaz.

Guest kullanıcısı **public** sunucu rolüne üyedir. Bir veritabanına Guest kullanıcısı ile erişebilmek için, veritabanına Guest kullanıcısı eklenmiş olmalıdır.

### **dbo KULLANICISI**

Veritabanı sahibinin yetkilerine sahiptir. SQL Server'da veritabanını oluşturan kişi veritabanının sahibidir. Veritabanındaki tüm izinler dbo kullanıcısına verilmiştir.

Veritabanındaki nesne sahiplikleri ve isimlendirmelerinin kullanıcılar ve dbo ile doğrudan ilişkisi vardır. Sysadmin sunucu rolü tarafından oluşturulan nesnelerin sahibi dbo olur. Sysadmin rolüne ait olmayan kullanıcıların oluşturduğu nesneler ise, nesneyi oluşturan kullanıcıya aittir. Bu farklılık, nesnenin isimlendirilmesini de etkiler. sysadmin sunucu rolüne ait kullanıcılar tarafından oluşturulan nesnelerin isimlendirmesi `dbo.Nesneİsmi` şeklinde olacakken, normal kullanıcıların oluşturacağı nesnelerin isimlendirmesi `kullaniciİsmi.Nesneİsmi` şeklinde olacaktır.

### **SYS VE INFORMATION\_SCHEMA KULLANICILARI**

SQL Server'da tüm sistem nesneleri **sys** ya da **INFORMATION\_SCHEMA** adlı şemaların içerisinde bulunur. Bu şemalar, her veritabanı için oluşturulmuş ve sadece master veritabanı içerisinde görünebilen özel şemalardır.

**INFORMATION\_SCHEMA** sorgulayarak veritabanı nesneleri hakkında bilgi alınabilir.

#### **Söz Dizimi:**

---

```
SELECT * FROM INFORMATION_SCHEMA.[ nesne_ismi ];
```

---

**INFORMATION\_SCHEMA** içerisinde bulunan nesneler aşağıdaki gibidir:

---

```
INFORMATION_SCHEMA.CHECK_CONSTRAINTS
INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE
INFORMATION_SCHEMA.COLUMN_PRIVILEGES
INFORMATION_SCHEMA.COLUMNS
INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE
INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS
INFORMATION_SCHEMA.DOMAINS
INFORMATION_SCHEMA.KEY_COLUMN_USAGE
INFORMATION_SCHEMA.PARAMETERS
INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
INFORMATION_SCHEMA.ROUTINE_COLUMNS
INFORMATION_SCHEMA.ROUTINES
INFORMATION_SCHEMA.SCHEMATA
INFORMATION_SCHEMA.TABLE_CONSTRAINTS
INFORMATION_SCHEMA.TABLE_PRIVILEGES
INFORMATION_SCHEMA.TABLES
INFORMATION_SCHEMA.VIEW_COLUMN_USAGE
INFORMATION_SCHEMA.VIEW_TABLE_USAGE
INFORMATION_SCHEMA.VIEWS
```

---

Şema içerisindeki tüm nesnelere ulaşmak için **SELECT \* FROM INFORMATION\_SCHEMA** yazdıktan sonra nokta (.) işaretini koymaz yeterlidir. **Management Studio**'nun **IntelliSense** özelliği açık ise, tüm şema içeriğini listeleyecektir.

Örnek kullanımı ise aşağıdaki gibidir:

---

```
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS;
```

---

	CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME	CHECK_CLAUSE
1	AdventureWorks2012	Production	CK_Product_ProductLine	(upper([ProductLine])='R' OR upper([ProductLine]...)
2	AdventureWorks2012	Production	CK_Product_Class	(upper([Class])='H' OR upper([Class])='M' OR upp...
3	AdventureWorks2012	Production	CK_Product_Style	(upper([Style])='U' OR upper([Style])='M' OR upp...
4	AdventureWorks2012	Production	CK_Product_SellEndDate	((SellEndDate)>=[SellStartDate] OR [SellEndDate]...
5	AdventureWorks2012	Production	CK_ProductCostHistory_EndDate	((EndDate)>=[Start Date] OR [EndDate] IS NULL)
6	AdventureWorks2012	Purchasing	CK_ShipMethod_ShipBase	((ShipBase)>(0.00))
7	AdventureWorks2012	Production	CK_ProductCostHistory_StandardCost	((StandardCost)>=(0.00))

**Sys** şeması içerisindeki veriler, **INFORMATION\_SCHEMA** şemasına göre daha fazladır. Bu verilere ulaşmak da önceki şema gibi kolaydır.

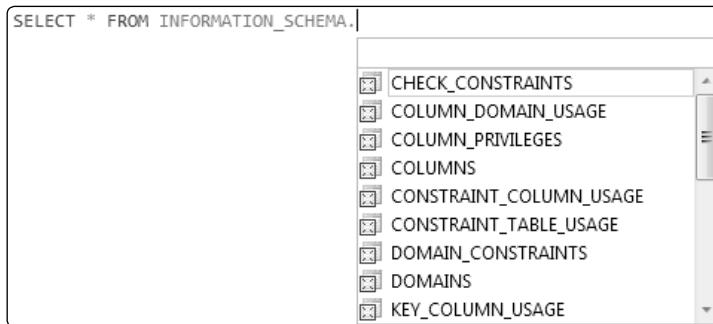
### Söz Dizimi:

---

```
SELECT * FROM sys.[ nesne_ismi ];
```

---

**Sys** şeması içerisindeki nesnelere ulaşabilmek için **SELECT \* FROM sys** yazdıktan sonra nokta(.) koymaz yeterlidir. **Management Studio IntelliSense** özelliği ile bu şema içerisindeki tüm nesneleri listeleyecektir.



Aktif olarak seçili olan veritabanında bulunan, tüm tablolardaki sütunları listeleyelim.

---

```
USE AdventureWorks
GO
SELECT * FROM sys.all_columns;
```

---

	object_id	name	column_id	system_type_id	user_type_id	max_length	precision	scale	collation_name	is_nullable	is_ansi_padded	is_rowguidcol
1	3	rsid	1	127	127	8	19	0	NULL	0	0	0
2	3	rsolid	2	56	56	4	10	0	NULL	0	0	0
3	3	hbcolid	3	56	56	4	10	0	NULL	0	0	0
4	3	rcmodified	4	127	127	8	19	0	NULL	0	0	0
5	3	ti	5	56	56	4	10	0	NULL	0	0	0
6	3	cid	6	56	56	4	10	0	NULL	0	0	0
7	3	ordkey	7	52	52	2	5	0	NULL	0	0	0

Sorgu sonucunda, binlerce sütun listelenecektir.

### SUNUCU OTURUMLARINI YÖNETMEK

SQL Server, Windows ile bütünleşik mimariye sahip olduğu için, iki farklı erişim güvenliği sunmaktadır. SQL Server'a Windows oturumları ile yetkili erişim sağlanabileceği gibi, **Mixed Security** (*Karmaşık Güvenlik*) seçimi yapıldıysa, her iki yöntem de kullanılabilir.

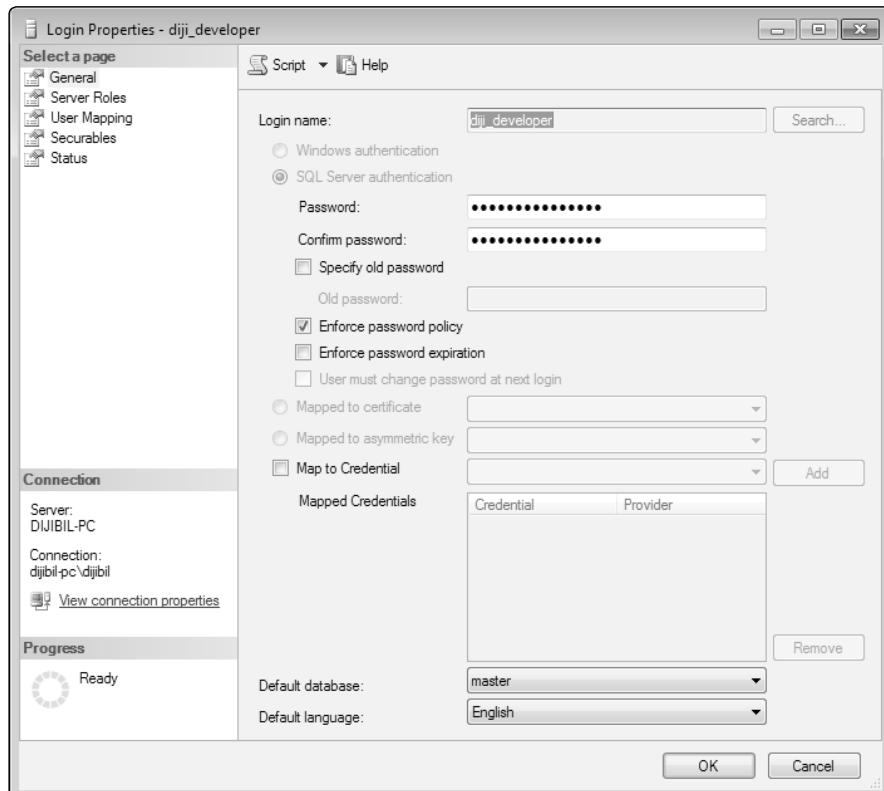
## OTURUMLARI GÖRÜNTÜLEMEK VE DÜZENLEMEK

SQL Server oturumlarını görüntülemek ve yönetmek için T-SQL ve Management Studio kullanılabılır.

Management Studio ile görsel olarak aşağıdaki gibi gerçekleştirilebilir.

- **Management Studio** ile bir sunucuya bağlandıktan sonra, **Object Explorer** panelinde **Security** klasöre girin.
- Oturumları görüntülemek için, **Security** içerisinde **Login** klasörüne girin.
- Login içerisinde, bir oturumun üzerine, farenin sağ düğmesiyle tıklayın ve özellikleri görebilmek için **Properties**'e tıklayın.

Seçilen oturumun **Properties** ekranı aşağıdaki gibi görünecektir.



**Properties** sayfasının ekran görüntüsündeki sayfaları inceleyelim.

- **General:** Oturumun özel ayarlarını gösterir. Kimlik doğrulama (değiştirilemez), şifre (değiştirilebilir), varsayılan dil ve varsayılan veritabanı ve şifre ile ilgili bazı seçim kutuları sunar.
- **Server Roles:** Sunucu rollerini listeler. Oturum için sunucu rolü ayarlarının yapılmasını sağlar.
- **User Mappings:** Seçili oturum tarafından erişilebilen veritabanlarını listeler. Atanmış veritabanı rollerini yönetmeyi sağlar.
- **Securables:** Seçili oturuma ait nesne izinlerini gösterir. Oturum için nesne izinlerini yönetmeyi sağlar.
- **Status:** Oturumun veritabanına bağlantısına izin verme ya da izni kaldırma ve oturumu aktif ve pasif etme gibi ayarların yapılmasını sağlar. Oturumun durumunu gösterir.



sa oturumunun **Properties** ekranında **Securable** sayfası yoktur.

Bir login hakkında bilgi almak için **sp\_helplogins** sistem **Stored Procedure**'ü kullanabilir.

#### Söz Dizimi:

---

```
sp_helplogins 'login_ismi'
cihan isimli login hakkında bilgi alalım.
EXEC sp_helplogins 'diji_developer';
```

---

	LoginName	SID	DefDBName	DefLangName	AUser	ARemote
1	diji_developer	0x5F71829396B4764186919053209E6C5C	master	us_english	yes	no
	LoginName	DBName	UserName	UserOrAlias		
1	diji_developer	AdventureWorks2012	AWtest	User		

Sorgu sonucunda, sorgulanınan login'in ismi, SID bilgisi, veritabanı ismi ve veritabanı dili gibi özet bazı bilgiler listeleniyor.

Oturum açmış kullanıcının, hangi sunucu rollerine ve Windows grubuna bağlı olduğunu öğrenmek için **sys.login\_token** kullanılır.

---

```
SELECT * FROM sys.login_token;
```

---

	principal_id	sid	name	type	usage
1	259	0x010500000000000515000000269B615A4D74262FD79D56...	dijibil-pc\dijibil	WINDOWS LOGIN	GRANT OR DENY
2	2	0x02	public	SERVER ROLE	GRANT OR DENY
3	3	0x03	sysadmin	SERVER ROLE	GRANT OR DENY
4	0	0x010500000000000515000000269B615A4D74262FD79D56...	dijibil-pc\None	WINDOWS GROUP	GRANT OR DENY
5	0	0x010100000000000000001000000000	\Everyone	WINDOWS GROUP	GRANT OR DENY
6	0	0x010500000000000515000000269B615A4D74262FD79D56...	dijibil-pc\HelpLibraryUpdaters	WINDOWS GROUP	GRANT OR DENY
7	0	0x01020000000000052000000020020000	BUILTIN\Administrators	WINDOWS GROUP	DENY ONLY

`sys.login_token` ile farklı sorgu birleştirmeleri yapılabilir. Örneğin; `sys.server_principals` ile birleşik sorgu oluşturalım.

---

```
SELECT LT.name,
       SP.type_desc
  FROM sys.login_token LT
    JOIN sys.server_principals SP
      ON LT.sid = SP.sid
 WHERE
   SP.type_desc = 'WINDOWS_LOGIN';
```

---

	name	type_desc
1	dijibil-pc\dijibil	WINDOWS_LOGIN

## KULLANICI, OTURUM ID'sİ VE PAROLA

Veritabanı kullanıcısı çok olan işletmelerde, kullanıcı ve oturum yönetiminde dikkat edilmesi gereken önemli hususlar vardır. Temel seviyede yönetilebilir veritabanı sistemine sahip işletmelerde ya da profesyonel BT ekibi ile veritabanı ve ağ yönetimi sağlanan işletmeler arasında, insan faktöründen dolayı çok büyük fark bulunmamaktadır.

Kullanıcıların oturumlarının parolalara bağlı olması, veritabanı güvenliğini artırma konusunda faydalı olsa da, çalışanların birçok sebepten dolayı, farklı erişim yetkilerine sahip veritabanı kullanıcılarını birbirlerinin kullanımına müsaade etmesi bazı güvenlik sorunlarına neden olur. Her kullanıcı (çalışan) ya da kullanıcı grubu için, farklı yetkilerin bulunması bir gereklilikdir. Ancak, örneğin bir çalışanın, işe gelmediği gün, bir başka çalışana, iş takibini yapmak ya da acil bir iş için gerekli veriyi elde edebilmek için, veritabanı erişim yetkileri veriyor olması bir güvenlik zafiyetidir. Çünkü kullanıcılar, veritabanı yöneticisi tarafından parola değişikliğine zorlanmadığı takdirde, sık sık parola değiştirmezler. Bu nedenle, zaman içerisinde, farklı kullanıcı bilgilerine sahip kullanıcılar nedeniyle, güvenlik yönetimi ve zafiyetlerin sebeplerini bulmak zorlaşacaktır.

Veri güvenliğine önem gösteren işletmeler, bu tür insan kaynaklı güvenlik zafiyetlerinin önüne geçebilmek için, işletmeden ayrılan bir personel olduğunda, parola kullanılan tüm kullanıcıların ya da personelin ayrıldığı birimde çalışan diğer personellerin, parolalarını değiştirmeye zorlar. Aynı şekilde, veritabanı ya da bilgisayar ağına karşı bir hacking saldırısı gerçekleştirildiğinde ya da şüphelenildiğinde, tüm personelin ve yetkili erişim gereken sistemlerin parolaları değiştirilir.

Profesyonel sistemlerde, çalışanlar belirli aralıklarla şifreleri değiştirmeye zorlanırlar. Ancak bir diğer sorun da, şifrelerin basit kombinasyonlardan oluşturulmasıdır. Brute Force, yani yazılımlar ile yapılan, deneme yanlış ile şifre bulma yöntemlerinin de saldırınlarda yaygın olarak kullanılıyor olması nedeniyle, şifrelerin basit değil, karmaşık ve farklı karakterler içeren kombinasyonlara sahip olması gereklidir.

### **PAROLANIN GEÇERLİLİK SÜRESİ**

Parola belirlendikten sonraki en önemli kavramdır. Bir işletmede çalışan herkesin bir kullanıcıya sahip olması, herkesin de bir parolaya sahip olduğu anlamına gelir. Parolalar, daha önce bahsettiğim çalışan güvenlik zafiyetlerinden dolayı, belirli aralıklarla değiştirilmeye zorlanmalıdır. Veritabanı yönetiminden sorumlu kullanıcılar, tüm kullanıcıların şifrelerini değiştirmeden veritabanına erişemeyecek şekilde güvenlik düzenlemesi yapar. Bu nedenle, kullanıcılar parolalarını değiştirmek zorunda kalırlar.

Parola geçerlilik süresinin uzun olmasının, hangi sorunlara neden olduğunu artık biliyor olmalısınız. Ancak parola geçerlilik süresinin kısa olması da bir o kadar sorunlu bir yöntemdir. Kullanıcılar, çok sık parola değiştirmeye zorlandığında, artık parolaları hatırlayamaz hale gelmekle birlikte, yeni parola olarak neyi belirleyeceğini bile düşünmez hale gelebilirler. 30 günde bir değiştirilen parola olduğunu varsayırsak, yılda 12 kez parola değiştirilir. Bir işletmede 2 aydır çalışan biri için sorun olmayabilir, ancak yıllardır çalışanlar için büyük sorundur. Zaman içerisinde kısa parola geçerlilik süresi nedeniyle, kullanıcılar basit parolalar oluşturmaya başlar. Yapılan araştırmalar şunu gösteriyor ki, bu durumda birçok kullanıcı, bulunduğu ay, yıl ya da birimin adını parola olarak belirleyecektir. Tavsiye edebileceğim parola geçerlilik süresi, 80 ile 160 gün arasında olmalıdır.

## **PAROLA UZUNLUĞU VE BİÇİMİ**

Parolayı deneme yanılma yolu ile bulmaya çalışan Brute Force yazılımları, parolaları farklı permütasyonlar ile bulmaya çalışırlar. Parola uzunluğu arttıkça, kombinasyon sayısı parolanın katları şeklinde artacaktır. Belirlenen parolaya eklenen her alfanümerik rakam, denenmesi gereken parolaların sayısını 36 kat (bazen daha fazla) artıracaktır. Sadece metinsel karakter yerine, nümerik ile birlikte metinsel karakter ve ek olarak özel karakterler (\*,-+^'{}\\ vb.) eklenmesiyle denenmesi gereken parola kombinasyonu sayısını milyonlara çıkarmak mümkündür.

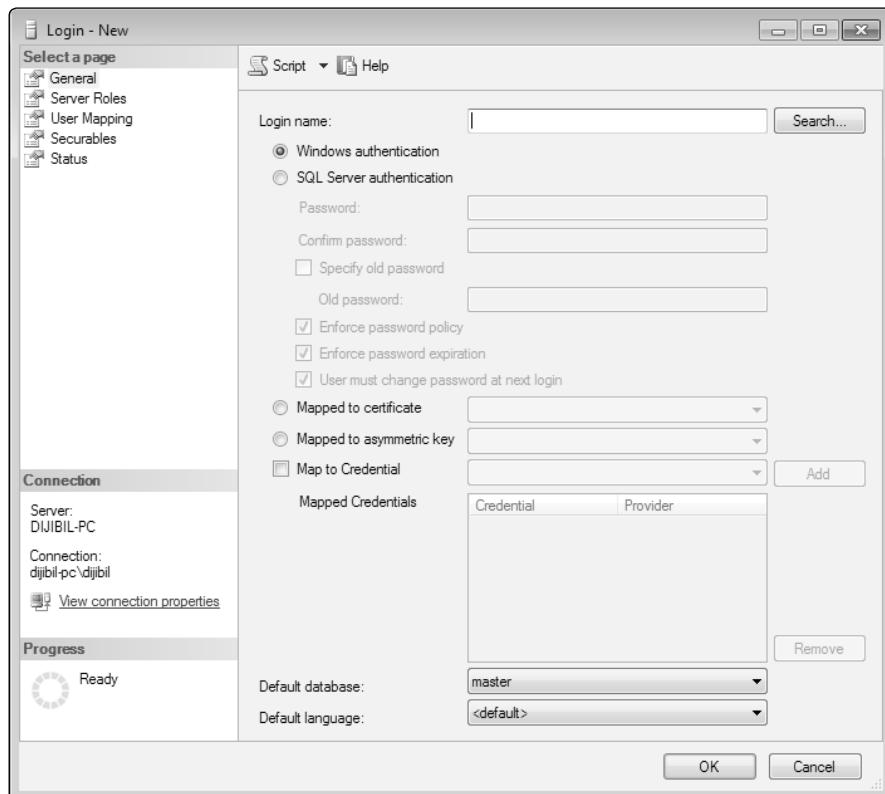
Güvenli parola oluşturmak önemlidir. Ancak oluşturulan parolanın hatırlanacak şekilde oluşturulması da ayrı bir öneme sahiptir.

## **OTURUMLAR OLUŞTURMAK**

SQL Server'da **Management Studio** ya da **Transact-SQL** kullanılarak yeni login oluşturulabilir. Windows ve **SQL Server Authentication** olmak üzere iki şekilde oturum oluşturulabilir. Windows oturumu oluşturmak için şifre belirlemeye gerek yoktur. Var olan bir Windows kullanıcı ya da grup hesabı kullanılabilir. Bunun için, **Login name** kısmında **Search** butonuna tıklayarak ilgili hesap seçilebilir. **SQL Server Authentication** modu seçili hale getirildiği takdirde, şifre ile ilgili alanlar düzenlenenebilir hale gelecektir. Bu alanları kullanarak, şifre değiştirebilir, bir sonraki girişte kullanıcı şifresini değiştirmeye zorlanabilir ve şifre için belirli bir gün sonra değişiklik yapma zorunluluğu getirilebilir.

Management Studio ile login oluşturmak için aşağıdaki adımları takip edin.

- **Management Studio**'da **Object Explorer** panelini açın.
- **Security** klasörü içerisinde **Logins** klasörüne sağ tıklayın ve **New Login...** butonuna tıklayın.
- Açılan pencerede, **Windows** ya da **SQL Server Authentication** ayarlarını yaparak **OK** butonu ile oluşturun.



Yeni oturum oluşturma seçenekinde birçok alan kolay ve anlaşılır şekildedir. Açıklanması gereken en önemli husus, **Windows Authentication** seçili olduğunda isimlendirme ile ilgili olan kısımdır.

Bir Windows hesabı için oturum oluşturuyorsanız, Windows Authentication seçenek düğmesini seçin. Login name kısmına, **ETKIALANI\kullaniciadi** formatında kullanıcı adını belirtin. Bu durumda, örneğin şu şekilde olacaktır; **DIJIBIL\gelistirici**. Ayrıca, **Active Directory** içerisinde etki alanını aramak istendiğinde **Search...** butonuna tıklanarak bu işlem gerçekleştirilebilir.

Mnagement Studio ile oturum oluşturmak kolaydır. Ancak, bazı durumlarda Transact-SQL ile programsal olarak oturum oluşturmak gerekir. Şimdi, Transact-SQL ile oturum oluşturmayı inceleyelim.

**Söz Dizimi:**


---

```

CREATE LOGIN login_name { WITH <option_list1> | FROM <sources> }

<option_list1> ::=

    PASSWORD = { 'password' | hashed_password HASHED } [ MUST_CHANGE
    ]
    [ , <option_list2> [ ,... ] ]

<option_list2> ::=

    SID = sid
    | DEFAULT_DATABASE = database
    | DEFAULT_LANGUAGE = language
    | CHECK_EXPIRATION = { ON | OFF}
    | CHECK_POLICY = { ON | OFF}
    | CREDENTIAL = credential_name

<sources> ::=

    WINDOWS [ WITH <windows_options>[ ,... ] ]
    | CERTIFICATE certname
    | ASYMMETRIC KEY asym_key_name

<windows_options> ::=

    DEFAULT_DATABASE = database
    | DEFAULT_LANGUAGE = language

```

---

Oturum söz dizimini kullanarak SQL oturumu, kimlik kartları ve etki alanına göre nasıl oturum oluşturulacağını inceleyelim.

**SQL OTURUMLARI OLUŞTURMAK**

```
CREATE LOGIN gelistirici WITH PASSWORD = '.._1=9(+%^+%dijibil';
```

**Kimlik kartları ile eşlenmiş SQL oturumu oluşturmak**

```
CREATE LOGIN gelistirici WITH PASSWORD = '.._1=9(+%^+%dijibil',
CREDENTIAL = DijibilCN;
```

**Etki alanı hesabından oturumlar oluşturmak**

```
CREATE LOGIN gelistirici FROM WINDOWS;
```

Oturum oluşturmanın temel kullanımı basit olmakla birlikte, söz dizimi genişletilerek bir çok ek özellik parametre olarak kullanılabilir.

## OTURUMLARI T-SQL İLE DÜZENLEMEK

Oturumları düzenlemek için Management Studio kullanılabilir. Bu yöntem kolay ve anlaşılırıdır. Ancak, programsal olarak düzenleme yapabilmek için biraz kod yazmak gereklidir.

T-SQL ile oturum düzenlemeleri için **ALTER LOGIN** kullanılır.

**ALTER** işlemi için farklı söz dizimi olsa da, yapı olarak **CREATE LOGIN** ile benzerdir. Bu nedenle tüm özelliklerine bu kitapta yer vermeyeceğiz.

**gelistirici** olan oturum adını **diji\_developer** olarak değiştirelim.

---

```
ALTER LOGIN diji_developer WITH NAME = diji_developer;
```

---

Oturum şifresini değiştirelim.

---

```
ALTER LOGIN diji_developer WITH PASSWORD = 'yeni_sifre-1()..,%11';
```

---

Kullanıcıdan, kullanıcı şifresini değiştirmesini isteyelim.

---

```
ALTER LOGIN diji_developer MUST_CHANGE;
```

---

Şifre ilkesini uygulayalım.

---

```
ALTER LOGIN diji_developer CHECK_POLICY = ON;
```

---

Şifre bitiş süresini uygulayalım.

---

```
ALTER LOGIN diji_developer CHECK_EXPIRATION = ON;
```

---

Oturum düzenleme için kullanılan **ALTER LOGIN** ifadesi, özel izinler ile kullanılabilir. Oturumları düzenlemek için **ALTER ANY LOGIN** iznine, kimlik kartları ile çalışan oturumlar için, **ALTER ANY CREDENTIAL** iznine sahip olunmalıdır.

## SUNUCU ERİŞİMİ VERMEK YA DA KALDIRMAK

Windows hesabı üzerindeki bir hesaba sunucunun **Database Engine** (*veritabanı motoru*)'ne erişim verilebilir ya da erişim kaldırılabilir.

**Management Studio** ile erişim işlemlerini yönetmek için aşağıdaki adımları takip edin.

- **Management Studio**'da **Object Explorer**'ı açın.
- **Object Explorer** panelindeki, **Security**'nin içerisinde **Logins** klasörünü açın.
- Bir oturum seçin ve sağ tıklayarak **Properties** ekranını açarak **Status** bölümüne girin.

Açılan ekranda, sunucuya erişim vermek için **Grant**, erişimi kaldırmak için **Deny** kullanılır.

Sunucuya erişimi kaldırmak, sadece Windows etki alanı hesabını kullanarak oturum açılmasını engeller. SQL Server oturumu kimliği ve şifresi bulunan kullanıcılar erişmeye devam eder.

Sunucuya erişimi yönetme işlemini Transact-SQL ile yapmak da kolaydır.

Sunucuya erişim vermek için **GRANT CONNECT** ifadesi kullanılır.

### Söz Dizimi:

---

```
GRANT CONNECT SQL TO login
```

---

Aşağıdaki gibi kullanılır.

---

```
GRANT CONNECT SQL TO diji_developer;
```

---

Sunucuya erişimi engellemek için **DENY CONNECT** ifadesi kullanılır.

### Söz Dizimi:

---

```
DENY CONNECT SQL TO login
```

---

Aşağıdaki gibi kullanılır.

---

```
DENY CONNECT SQL TO diji_developer;
```

---

**DENY** ve **GRANT** sorguları master veritabanı üzerinde çalıştırılmalıdır.

## **OTURUMLARI ETKİNLEŞTİRMEK, DEVRE DİŞİ BIRAKMAK VE KİLİDİNİ KALDIRMAK**

SQL Server oturumlarını etkinleştirilebilir ya da devre dışı bırakılabilir. Bu işlem kullanıcı tarafından Management Studio ya da Transact-SQL ile yapılabilir. SQL Server, bazı durumlarda kendisi de otomatik olarak sistemi kilitlemek için etkinleştirme/devre dışı bırakma yöntemini kullanabilir. Örneğin; oturum şifresinin süresi dolduğunda devre dışı bırakılarak oturum kilitlenir.

Bu işlemleri yönetmek için iki şekilde de, nasıl yapılacağını inceleyeceğiz.

Management Studio ile etkinleştirme/devre dışı bırakma özelliklerini yönetmek için aşağıdaki adımları izleyin.

- **Management Studio** ekranında, **Object Explorer** paneline girin.
- **Security** klasörü içerisinde **Logins** klasörüne girin.
- Bir login seçerek üzerine sağ tıklayın ve oturum özelliklerini görmek için **Properties** butonuna tıklayın.
- Sol menüdeki **Status** kısmını açın.
- Etkinleştirmek için;  
**Login** altında **Enabled**'ı seçin.
- Devre dışı bırakmak için;  
**Login** altında **Disabled**'ı seçin.

Transact-SQL ile hesap etkinleştirme, devre dışı bırakma ve kilidini kaldırmayı inceleyelim.

### **Söz Dizimi:**

---

```
ALTER LOGIN login_ismi DISABLE | ENABLE | UNLOCK
```

---

diji\_developer oturumunu devre dışı bırakalım.

---

```
ALTER LOGIN diji_developer DISABLE;
```

---

diji\_developer oturumunu etkinleştirelim.

---

```
ALTER LOGIN diji_developer ENABLE;
```

---

diji\_developer oturumunun kilidini açalım.

---

```
ALTER LOGIN diji_developer WITH PASSWORD = '123456' UNLOCK;
```

---

## ŞİFRELERİ DEĞİŞTİRMEK

Oturum şifrelerini değiştirmek sık gerçekleştirilen bir işlemdir. Bu işlem de Management Studio ile yapılabileceği gibi T-SQL ile de yapılabilir.

Management Studio ile görsel olarak şifre değiştirmek için daha önceki anlatılan yöntemleri izleyerek Security içerisindeki Logins klasöründe bir oturuma sağ tıklayıp, Properties'e giriş yapın ve yukarıdaki şifre alanlarını doldurarak aşağıdaki OK butonuna basın. Artık şifreniz değiştirilmiş olacaktır. Şifreleri karmaşık ve güçlü oluşturmanız önerilir. Oturum şifresini değiştirmek için, gerekli T-SQL ifadesini kullanmıştık. Şimdi özetle tekrar belirtelim.

T-SQL ile oturum şifresini değiştirmek.

---

```
ALTER LOGIN diji_developer WITH PASSWORD = 'yeni_sifre11-2&5%';
```

---

Kullanıcıdan, kullanıcı şifresini değiştirmesini isteyelim.

---

```
ALTER LOGIN diji_developer MUST_CHANGE;
```

---

## OTURUMLARI KALDIRMAK

Bir kullanıcı ile veritabanına erişen çalışan işten ayrıldığında ya da bir şekilde bu görevden, dolayısıyla veritabanı yetkilerinden ayrılması gerektiğinde, oturum silinmelidir.

Oturum silme işlemi, **Management Studio**'da bir oturuma sağ tıklanarak ve **Delete** butonu kullanılarak gerçekleştirilebilir. Ayrıca, T-SQL kullanılarak da oturum silinebilir.

### Söz Dizimi:

---

```
DROP LOGIN login_ismi
```

---

Biz, T-SQL kullanarak oturumu silelim.

---

```
DROP LOGIN diji_developer;
```

---

**Management Studio** içerisinde, **Object Explorer** paneline girin. Buradaki **Security** klasöründe bulunan **Logins** klasörü içerisinde, artık **diji\_developer** isminde bir oturum bulunmadığını görebilirsiniz.

## İZİNLER

Veritabanında, nesneler üzerinde işlemler gerçekleştirebilmek için, iznlere ihtiyaç vardır. Veritabanında bir işlem yapılabilmesi için, öncelikle bu işlemi gerçekleştirebilecek erişim iznine sahip olunmalıdır.

Veritabanında izinleri, veritabanı sahibi, **sysadmin** üyeleri ve **securityadmin** üyeleri atayabilirler. Bu izinler şunlardır;

- **GRANT**: Belirlenen görevi gerçekleştirmeye izni verir. Rollerle çalışırken, tüm rol üyeleri izni devralır.
- **REVOKE**: Verilmiş olan **GRANT** iznini geri alır. Ancak, kullanıcının ya da rolün bir görevi gerçekleştirmesini engellemez. Sadece verilen izni geri alır. Kullanıcı ya da rol, başka bir rolün **GRANT** iznini devralabilir.
- **DENY**: Bir izni iptal eder. **REVOKE**'den farkı, **DENY** izinleri rolün izni devralmasını önler. Yani açıkça bir izin yasaklanır. **DENY**, diğer tüm **GRANT** izinlerinden yüksek önceliğe sahiptir.

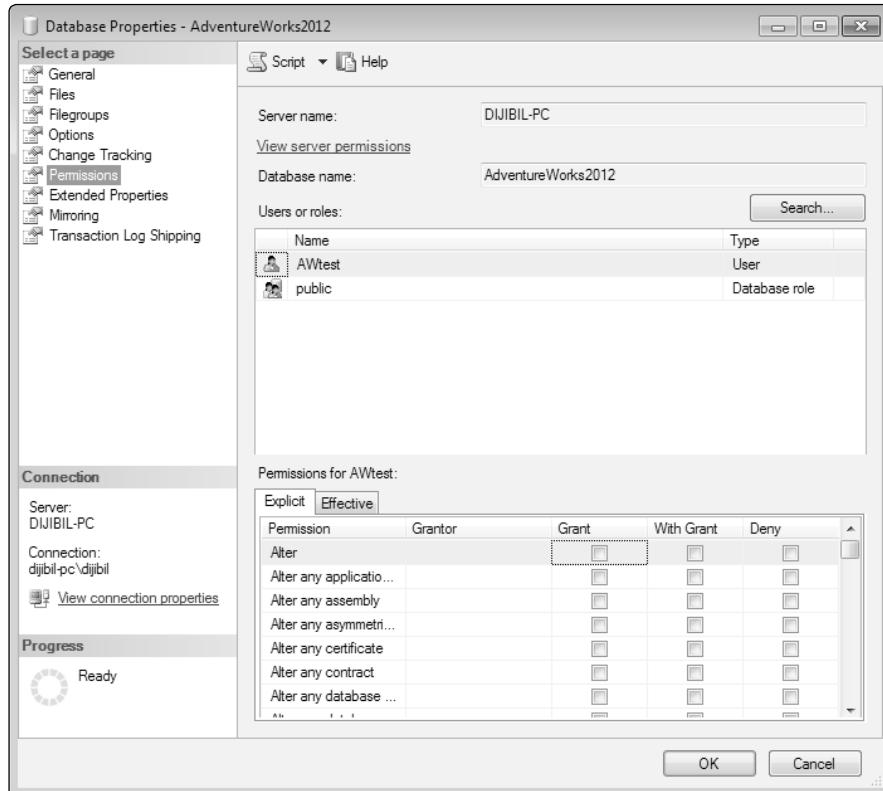
## İFADE İZİNLERİ

Veritabanında, **CREATE DATABASE**, **CREATE TABLE** ya da **BACKUP DATABASE** gibi DDL ifadelerini çalıştırmak için izinler kullanılabilir. Bu izinler verilebilir, geri alınabilir ya da iptal edilebilir.

İzinleri yönetmek için **Management Studio**'nun yeteneklerinden yararlanılabilir. Aynı zamanda, T-SQL ile de kolaylıkla yapılabilir.

**Management Studio** ile izinleri yönetmek için aşağıdaki adımları izleyin.

- **Management Studio’da Object Explorer** panelini açın.
- **Object Explorer’dan Databases** klasörüne girin.
- Bir veritabanı seçin ve veritabanı adına sağ tıklayarak **Properties** butonuna tıklayın.
- Açılan penceredeki sol menüden, **Permissions**’a tıklayın.



Bu işlemler genel olarak Management Studio ile gerçekleştirilir. Ancak, bazen T-SQL ile çözüm bulmanız gereklidir. T-SQL ile izinler atamayı ve yönetmeyi inceleyelim.

**GRANT** ile izin atama ifadesini inceleyelim.

**AdventureWorks** veritabanı kullanıcılarından **AWtest**'e, **Production.Product** tablosu için **SELECT** izni verelim.

---

```
GRANT SELECT  
ON Production.Product  
TO AWtest;
```

---

**AWtest** kullanıcıı için **CREATE TABLE** izni verelim.

---

```
GRANT CREATE TABLE  
TO AWtest;
```

---

**AWtest** kullanıcıına **Production.Product** tablosu üzerinde **INSERT**, **UPDATE** ve **DELETE** komutlarını kullanma izni verelim.

---

```
GRANT INSERT, UPDATE, DELETE  
ON Production.Product  
TO AWtest;
```

---

**REVOKE** ile verilen izinleri geri alalım.

**AdventureWorks** veritabanı kullanıcılarından **AWtest**'e, **Production.Product** tablosu için verdigimiz **SELECT** iznini geri alalım.

---

```
REVOKE SELECT  
ON Production.Product  
TO AWtest;
```

---

**AWtest** kullanıcıı için verdigimiz **CREATE TABLE** iznini geri alalım.

---

```
REVOKE CREATE TABLE  
TO AWtest;
```

---

**AWtest** kullanıcıına, **Production.Product** tablosu üzerinde **INSERT**, **UPDATE** ve **DELETE** komutlarının kullanılması için verilen izni geri alalım.

---

```
REVOKE INSERT, UPDATE, DELETE  
ON Production.Product  
TO AWtest;
```

---

**DENY** ile verilen izinleri iptal edelim.

**AdventureWorks** veritabanı kullanıcılarından **AWtest**'e, **Production.Product** tablosu için verdığımız **SELECT** iznini iptal edelim.

---

```
DENY SELECT  
ON Production.Product  
TO AWtest;
```

---

**AWtest** kullanıcısı için verdığımız **CREATE TABLE** iznini iptal edelim.

---

```
DENY CREATE TABLE  
TO AWtest;
```

---

**AWtest** kullanıcısına, **Production.Product** tablosu üzerinde **INSERT**, **UPDATE** ve **DELETE** komutlarının kullanılması için verilen izni iptal edelim.

---

```
DENY INSERT, UPDATE, DELETE  
ON Production.Product  
TO AWtest;
```

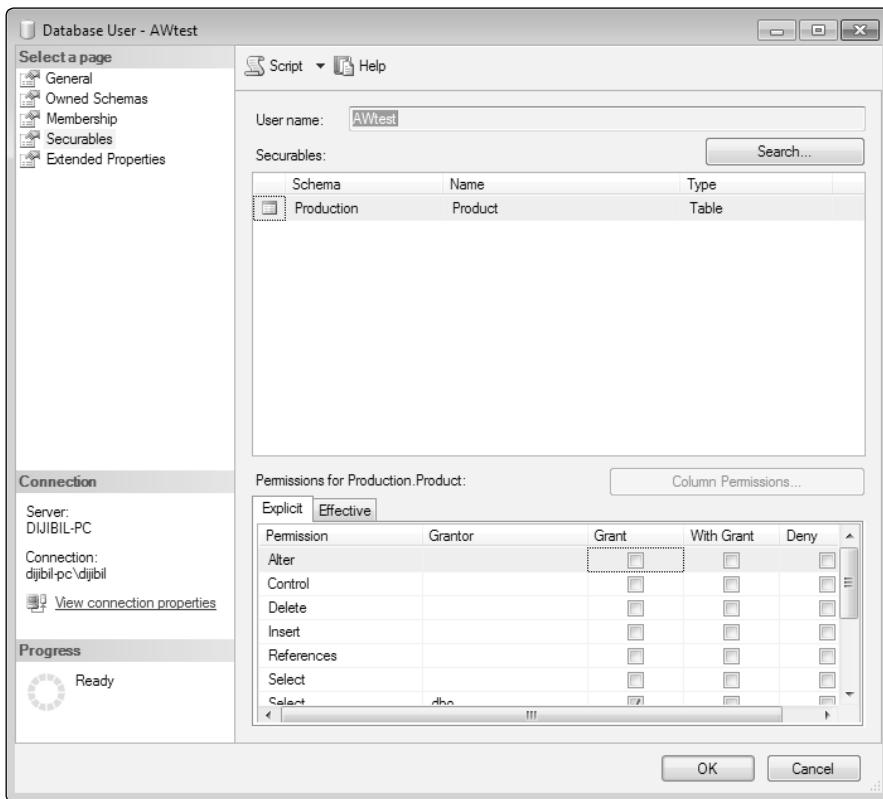
---

## NESNE İZİNLERİ

Tablo ve view, Stored Procedure gibi nesnelere verilen izinlere denir. Bu nesnelere **SELECT**, **INSERT**, **UPDATE** ve **DELETE** izinleri dahildir.

**Management Studio** ile hızlı bir şekilde nesne izinleri atanabilir.

- **Management Studio**'da **Object Explorer** panelini açın.
- **Databases** klasörüne içerisinde bir veritabanı seçerek düğümünü açın.
- **Security** ve **Users** klasörlerini açın.
- Ayarlarını değiştirmek istediğiniz kullanıcıyı fare ile çift tıklayın.



- Açılan pencerede **Search...** butonuna tıklayarak **Add Objects** menüsünü açın. Bu ekranda farklı yollar ile nesnelere ulaşarak gerekli nesne izinleri atanabilir.

## ROLLER

Windows güvenlik grupları gibi bir kullanıcı grubuna izinler vermek için kullanılırlar.

- Sunucu Rollerleri:** Sunucu düzeyinde uygulanan rollerdir.
- Veritabanı Rollerleri:** Veritabanı düzeyinde uygulanan rollerdir.

## SUNUCU ROLLERİ

Sunucunun birçok özelliğini yönetmek için oluşturulmuş rollerdir. Oturum ve kullanıcılarla sunucuya çeşitli kategoriler ile yönetme imkanı vermek için kullanılır. Bir oturum, bir grup üyesi yapıldığında, bu oturumu kullanan kullanıcılar, rol tarafından izin verilen her görevi gerçekleştirebilirler.

Sunucu rollerini, yukarıdan aşağı doğru listeleyerek inceleyelim.

- **sysadmin:** SQL Server'ın tam yönetimi için hazırlanan en yetkili sunucu rolüdür. Bu rolün üyeleri, SQL Server üzerinde tüm işlemleri gerçekleştirebilir.
- **setupadmin:** Bağlı sunucuları yönetmek ve Stored Procedure'leri başlatması gereken kullanıcılar için tasarlanmıştır.
- **serveradmin:** Sunucu çapında temel seviye yönetim işlemlerini gerçekleştirmek için kullanılabilir. Sunucu yapılandırma seçeneklerin ayarlanması sağlarlar. Şu işlemleri gerçekleştirebilirler;

**SHUTDOWN**

**RECONFIGURE**

**DBCC FREEPROCCACHE**

**sp\_configure**

**sp\_tableoption**

**sp\_fulltext\_service**

- **securityadmin:** Sunucu güvenliğini yönetmesi gereken kullanıcılar için tasarlanmıştır. Bu role sahip kullanıcılar, oturumları, veritabanı izinlerini oluşturma ve yönetme, şifreleri sıfırlama, hata günlüklerini okuma, sunucu ve veritabanı düzeyli izinleri yönetme işlemlerini gerçekleştirebilirler. Şu işlemleri gerçekleştirebilirler;

**CREATE LOGIN**

**ALTER LOGIN**

**DROP LOGIN**

**GRANT CONNECT**

**DENY CONNECT**

- **processadmin:** SQL Server işlemlerini yönetmesi gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri işlemleri sonlandırabilir.

- **diskadmin:** Disk dosyalarını yönetmesi gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, **sp\_addumpdevice** ile **sp\_dropdevice** kullanabilirler.

- **dbcreator:** Veritabanları oluşturma, düzenleme, kaldırma ve geri yükleme işlemlerini gerçekleştirecek kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, şu işlemleri gerçekleştirebilirler;

**CREATE DATABASE**  
**ALTER DATABASE**  
**DROP DATABASE**  
**EXTEND DATABASE**  
**RESTORE DATABASE**  
**RESTORE LOG**

- **bulkadmin:** Veritabanına **BULK** veri ekleme işlemleriini gerçekleştirecek kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, **BULK INSERT** ifadesi çalıştırabilirler.

## **VERİTABANI ROLLERİ**

Veritabanı düzeyinde izinler atamak istendiğinde veritabanı rolleri kullanılabilir.

Üç çeşit veritabanı rolü vardır. Bunlar;

### **KULLANICI TANIMLI STANDART ROLLER**

Benzersiz, izin ve haklarla roller oluşturulmasını sağlar. Kullanıcıların, mantıksal olarak gruplandırılmış, belirli izinlerle oluşturulan bir rol ile yönetilmesini sağlar. Çok kullanılan bir örnek; normal kullanıcılar veritabanı üzerinde sadece **SELECT, INSERT ve UPDATE** sorgusu ile hazırlanacak bazı işlemler gerçekleştirir. Sadece bu üç tip sorugu çalıştırabilecek ve diğer komutlara izin vermeyecek **Users** adında bir rol oluşturulabilir.

### **KULLANICI TANIMLI UYGULAMA ROLLERİ**

Belirli uygulamalar için şifre korumalı roller oluşturulmasını sağlar. Örneğin; bir uygulama veritabanına bağlanır ve kendisi için oluşturulan uygulama rolünün yetkisi dahilinde işlemler gerçekleştirilebilir. Bir uygulama rolüne, diğer roller atanamaz.

### **ÖNCEDEN TANIMLI VERİTABANI ROLLERİ**

SQL Server tarafından önceden tanımlanmış rollerdir. Bu roller değiştirilemeyen izinlere sahiptir. Birden çok role tek bir oturum atanabilecek şekilde yönetilebilirler.

Bu roller;

- **public:** Tüm veritabanı kullanıcıları için, minimum izin ve haklara sahip varsayılan roldür. Public rolü haricinde, kullanıcıya farklı rol ve izinler atanabilir.
- **db\_accessadmin:** Bir veritabanına oturum eklemesi ya da var olan oturumları kaldırması gereken kullanıcılar için tasarlanmıştır.
- **db\_backupoperator:** Veritabanı yedekleme işlemlerini yöneten kullanıcılar için tasarlanmıştır.
- **db\_datareader:** Veritabanındaki veriyi görmesi gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, veritabanındaki herhangi bir tablodan veri alabilir.
- **db\_datawriter:** Veritabanındaki bir tabloya veri ekleme ya da güncelleme işlemi gerçeklestirmesi gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, seçili veritabanındaki tüm nesneler üzerinde aşağıdaki işlemleri gerçekleştirebilir.

**INSERT**

**UPDATE**

**DELETE**

- **db\_ddladmin:** Adından da anlaşılacağı gibi, **DDL** (*Data Definition Language*) ile ilgili işlemler gerçeklestirmesi gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, aşağıdaki işlemleri gerçekleştirebilir;

**GRANT**

**REVOKE**

**DENY**

Tüm DDL İşlemleri

- **db\_denydatareader:** Bir oturumun veri erişimini kısıtlamak için tasarlanmıştır. Bu rolün üyeleri, veritabanının kullanıcı tablolarındaki veriyi okuyamazlar.
- **db\_denydatawriter:** Bir oturumun veri ekleme, değiştirme ve silme işlemlerini kısıtlamak için tasarlanmıştır. Bu rolün üyeleri, veritabanı kullanıcı tablolarındaki veri üzerinde aşağıdaki işlemleri gerçekleştiremezler.

**INSERT**

**UPDATE**

**DELETE**

- **db\_owner:** Veritabanı üzerinde tam yetkili olması gereken kullanıcılar için tasarlanmıştır. Bu rolün üyeleri, tüm veritabanı işlemlerini gerçekleştirebilirler.
- **db\_securityadmin:** Veritabanı izinlerini, nesne sahipliği ve rolleri, yani veritabanı güvenlik ayarlarını yönetmesi gereken kullanıcılar için tasarlanmıştır.
- **dbm\_monitor:** Veritabanı ikizlemenin mevcut durumunu izlemesi gereken kullanıcılar için tasarlanmıştır.

## SUNUCU ROLLERİNİ YAPILANDIRMAK

Sunucu rollerini rol ya da oturumlar ile yönetmek mümkündür.

### OTURUM İLE ROLLERİ ATAMAK

Bir oturum için rolleri atama işlemi Management Studio ve T-SQL ile yapılabilir. İki yöntemin de kullanılması gereken özel durumlar olabilir.

Bu işlemi Management Studio ile gerçekleştirmek için aşağıdaki adımları izleyin.

Önceki örneklerde anlatıldığı gibi **Logins** ekranına gelin ve bir oturuma sağ tıklayarak **Properties** butonuna tıklayın. Açılan ekrandaki seçim kutucuklarını seçerek ya da seçili olanları kaldırarak gerekli atamaları yaptıktan sonra **OK** butonu ile değişiklikleri kaydedin.

Aynı işlem Transact-SQL ile gerçekleştirilebilir.

#### Söz Dizimi:

---

```
sp_addsrvrolemember [@loginame =] login, [@rolename =] role
```

---

**diji\_developer** oturumunu sysadmin rolüne ekleyelim.

---

```
EXEC sp_addsrvrolemember diji_developer, sysadmin;
```

---

**diji\_developer** oturumunu sysadmin rolünden kaldırıyalım.

#### Söz Dizimi:

---

```
sp_dropsrvrolemember [@loginame =] login, [@rolename =] role
```

---

`diji_developer` Oturumunu `sysadmin` rolünden kaldıralım.

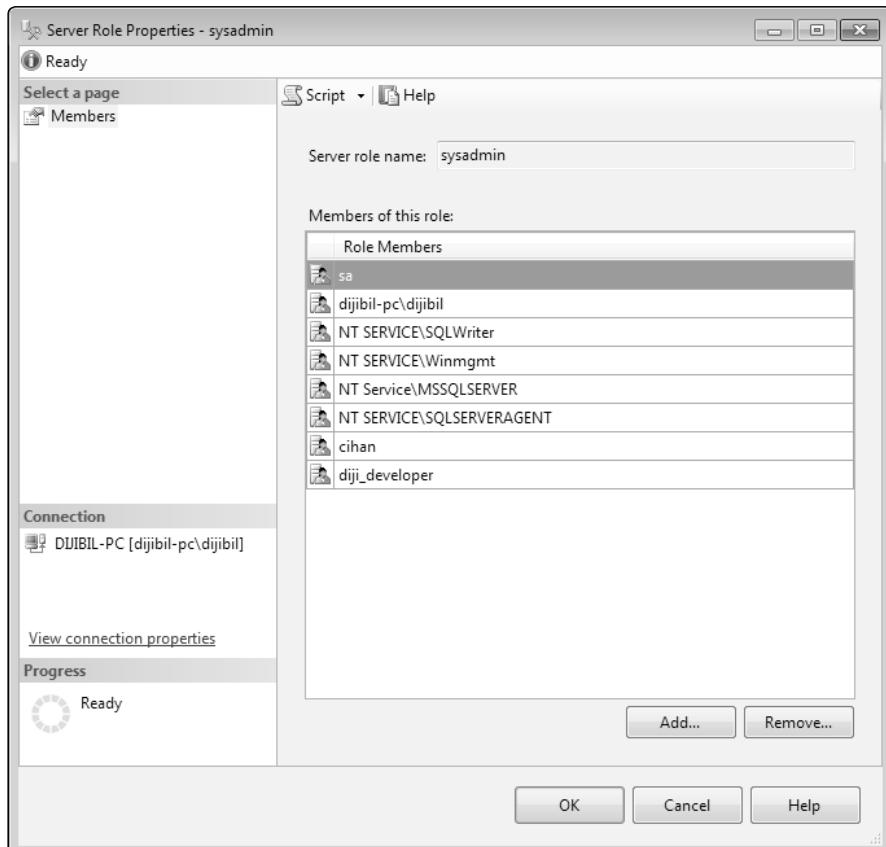
---

```
EXEC sp_dropsrvrolemember diji_developer, sysadmin;
```

---

## BİRDEN ÇOK OTURUMA ROLLER ATAMAK

Birden çok oturuma roller atamak için **Management Studio** kullanılabilir. Bunun için önceki örneklerdeki benzer yolu takip edeceğiz. **Management Studio** içerisinde **Object Explorer**'ı açın. Daha sonra **Security** klasörü içerisinde **Server Roles** klasörünü açın. Klasör içerisindeki rollerden herhangi birine fare ile çift tıklayın.



Açılan pencerede, **Add** ya da **Remove** butonlarını kullanarak, yeni ekleme ya da çıkarma işlemleri yapılabilir.

# PERFORMANS VE SORGU OPTİMİZASYONU

19

Bilgisayar dünyasında en karmaşık ve sinir uçları belli olmayan iki konudan biri performans, diğerinin güvenlik kavramıdır. SQL Server mimarisini, birçok bölümde, farklı katmanlarını inceleyerek irdeledik. Temel ve ileri seviye T-SQL programlama konularını tüm detaylarıyla inceledik. Ancak, performans kavramını tam olarak kavrayabilmek için bu bilgiler yeterli değildir. Bunun nedeni; performans kavramının farklı açılardan irdelenmesi ve optimize edilerek yönetilmesi gereken bir çok farklı iş katmanına sahip olmasıdır.

Bir veritabanı performansı artırma işlemine en az aşağıdaki açılardan bakmak gereklidir.

- Müşterinin performans denilince anladığı ve istediği nedir?
- Veritabanı doğru tasarlandı mı?
- Sistem donanım performansı yeterli mi? (İşlemci, Ram, HDD vb.)
- T-SQL bazında nesne ve sorgular performansa uygun şekilde kullanıldı mı?
- Sistem ağ trafiği veri iletişim performansı açısından uygun mu?
- Veritabanı dağıtık mimariye uygun tasarlandı mı?
- Bakım ve yedeklemeler performansı dengeleyecek şekilde mi yapılıyor?
- Veritabanını kullanan istemcilerin sorgu performans durumu nedir?

Bu sorular sadece akla gelen en temel konular içindir. Her bir sorunun detayına girildiğinde birer kitap halini alabilir.

Biz SQL Server'da performans konusunu derinlemesine incelemek yerine, genel kapsamda neler yapabileceğimize bakacağız. Teorik anlamda zihinlerde bir başlangıç noktası ve kavramsal model oluşturmak için bu bölümde çeşitli konulara değineceğiz.

## PERFORMANS AYARI NE ZAMAN YAPILMALIDIR?

Performans ayarı projeye başlangıç tarihidir. Dikkat ederseniz T-SQL geliştirmeye başlangıç değil, projeye başlangıç! Performans durumu, projenin büyüklüğü ve hedefleri genellikle doğru orantılıdır. Küçük çaplı projelerde de performans gereklidir. Ancak, bu gereklilik çok kritik değildir. Büyük veritabanı mimarilerinde en ufak performans sorunu katlanarak büyük bir soruna dönüşür. Bunun nedeni; büyük sistemlerin daha yüksek sorgu sayılarına ve kullanıcılarla sahip olmasıdır.

Bir projeye başlandığında gereklilikler zaten hazırlanmış olduğu varsayılar. Bu durumda, bu veritabanının ne için tasarlandığı biliniyor. Öncelik sırası veritabanı tasarımına aittir. Donanım performansının, gerçek veri yükü ile çalışmaya başlayan kadar normal olması yeterlidir. Veritabanı tasarımını, geliştirmeden önceki en önemli unsurdur. Eğer veritabanı tasarımında kritik hatalar yapılsa, geliştirme sürecinde sürekli yama yapmak ve düzenlemeler gerçekleştirmek için geri dönmek ve mimari değiştirilmek zorunda kalınabilir. Ayrıca, ilişkisel veritabanlarında veritabanı tasarımını, doğrudan sorgu performansı üzerinde etkilidir.

Doğu tasarlanan veritabanında, artık T-SQL geliştirme mimarisi açısından performanslı sorgular geliştirilmelidir. İndeks ve constraint'ler doğru kullanılmalı, veri tekrarından kaçınılmalıdır. Bu aşamada gerçekleştirilmesi gereken T-SQL performans konularında çok detaylara girmeyeceğiz. Ancak, T-SQL'de kullanılan tüm sorgular performanslı değildir. Veri istatistiklerini takip ederek bu konuda tecrübe edinmeniz ve düzenli olarak performans iyileştirmeleri yapmanız gereklidir.

Veritabanı performans iyileştirmeleri projenin başlangıcıyla başlar ve proje tamamıyla kullanımından kalkmadan bitmez.

## DONANIM

Daha önce proje analizi ve yönetim tecrübe olmayanlar için donanım kısmındaki keskin uçları anlamak kolay olmayabilir. Aslında bir geliştirici için çok kritik ve uzmanlık gerektiren bir konu değildir. Fakat genel anlamda bir proje için hayatı duruma sahiptir.

Veritabanı ve yazılımlar için donanım, para-kasa ilişkisine benzer. Veri, yani veritabanı ya da yazılım para ise; donanım da bunları koruyacak ve hasarsız saklanmasılığını sağlayacak kasadır.

Nesneleştirmeyi daha da ilginç hale getirirsek, donanım insan vücutu ise, veritabanı bu vücutun beynidir. Beynin performanslı çalışması ve güvenliğinin sağlanabilmesi için vücutun güçlü ve sağlıklı olması gereklidir.

Donanım yatırımları projeye mali açıdan büyük yük oluşturabilir. Ancak depoladığınız verinin fiyatı nedir?

Bir online alışveriş sitesini senaryolaştıralım. Binlerce ürüne ve günlük on binlerce kullanıcı girişine, binlerce satış grafiğine sahipsiniz. Yazılım ve veritabanı sorunsuz çalışıyor. Ancak donanım bu kapasiteyi kaldırılamayacak özelliklere sahipse, yazılım ve veritabanı göçme senaryoları gerçekleşebilir. Sisteme erişim sağlanamayabilir ve bu sorunların alışveriş sistemine maliyeti hem maddi hem de müşterilerin güvenini kaybetmek gibi paha biçilemez sonuçlara yol açabilir.

İş bu kadariyla da bitmez. Veritabanı ve yazılımı garantiye almak bazen farklı sunucular üzerinden hizmet verilmesi gereklidir. Bu durum donanım maliyetlerini tabii ki artıracaktır. Örneğin; bir video sitesinin de yazılım ve veritabanı gereksinimi yüksektir. Kullanıcı bilgileri, özel veriler ve video bilgileri tamamen veritabanında tutulur. Video işlemleri yüksek trafik ve donanım tüketirler. Birçok sunucu hatasına da yol açabilirler. Bu tür durumlarda, veritabanı ve yazılımı farklı bir donanım üzerinde barındırmanız gerekecektir.

Ayrıca sistemin lokal mi, yoksa global bir ağ içerisinde mi barındırılması konusu da önemlidir. Bunun için, sistemin sunucu olarak kullanılıp kullanılmayacağına daha önceden belirlenmelidir. Veritabanı sunucuları şirket bünyesinde bir sunucu odasında tutulabileceği gibi, özel bir veri merkezinde de tutulabilir. Bu iki seçenekten doğru olanı seçmek önemlidir. Sadece firma içerisindeki bir

veritabanını barındıracak bir sunucunun firma bünyesinde tutulması doğru olabilir. Ancak bir alışveriş sistemi örneğinde, global bir ağ oluşması nedeniyle, yüksek trafik ve veri kapasitesi, güvenlik ve elektrik kesintisi gibi riskler nedeniyle veri merkezinde barındırma fikri daha doğru olacaktır.

Özetlemek gerekirse, donanımın belirlenmesi için şu sorular sorulabilir.

- Sunucu üzerinde işlem yoğunluğu I/O mu, yoksa işlemci üzerinde mi gerçekleşecek?
- Sistem çökerse, risklerim neler ve veri kaybı senaryolarına hazır mıyım?
- Öncelik performans mı, yoksa verinin güvenliği mi?
- Donanım, bir sunucu olarak mı kullanılacak?
- Sunucu ise, bu sunucu bulunulan yerde mi yoksa bir veri merkezinde mi bulunacak?
- Sunucu uzakta ise, en kritik göçme senaryolarında müdahale edilecek teknik ve eleman var mı?
- Veritabanının bulunduğu sunucu tamamen çökerse aynı anda aynı veriye sahip yeni bir sunucunun devreye girmesini sağlayacak aynalama işlemi gerçekleştirildi mi?

## I/O VE CPU YOĞUNLUĞU VE RAM KULLANIMI

SQL Server ya da herhangi bir veritabanının en yoğun kullandığı donanım unsuru işlemcidir. Bir makinede birden fazla yoğun işlem gerçekleştiren yazılım çalışıyorsa, bunların her biri işlemciyi daha etkin kullanmak için rekabet edeceklerdir. Günümüzde işlemci maddi açıdan daha ucuz ve teknik olarak daha gelişmiş özelliklere sahip olduğundan, SQL Server çalışan makinede birden fazla işlemcinin olması bir şart olarak görülmelidir.

Tek işlemci çalışan bir sistemde, durum "SQL Server ve diğer yazılımlar" olarak düşünülmelidir. Bir sistemde sadece SQL Server çalışmaz. Geliştirme ortamı bile olsa, bir sistemde en az iki işlemci olmalıdır. En azından, diğer yazılım ve işlemlerin SQL Server'ın kullanacağı işlemci gücünü tüketmemesi sağlanmış olur.

SQL Server farklı sürümlerde farklı işlemci kullanım yeteneklerine sahiptir. Standard Edition sürümü dört işlemciyi desteklerken, Enterprise Edition sürümü donanımınızın en üst sınırına kadar işlemci desteği sağlar. Genel olarak geliştirme ortamı ve ücretsiz seçenek olan Express Edition ise tek işlemci kullanabilir. Tek işlemci kullanması durumunu önemsenmelidir. Makinede bir işlemci varsa, Express Edition, maksimum kullanabileceği işlemci gücünü, tüm sistemdeki yazılım ve servisler ile paylaşmış olur. İki işlemci olursa birini SQL Server kullanırken diğer sistem yazılımları ikinci işlemciyi kullanabilir.

CPU'nun ayrılmaz parçası olarak RAM kullanımını düşünebiliriz. RAM, bir veritabanı için olmazsa olmaz ve yüksek boyutlarda olması önerilen gereksinimdir.

Veritabanı boyutu, kullanıcı sayısı, veri büyülüğu, sorguların karmaşıklığı (gruplama, join'ler vb.) gibi farklı parametrelere göre RAM ihtiyacı belirlenir.

Bir kullanıcının sisteme bağlanması yaklaşık 24K RAM kullanır. Her sorgunun sonucunda alınan veri RAM'de bir alan kaplar. Sistem açık kaldığı ve çalıştığı sürece bu kullanım sürekli artar. Veritabanı üzerinde mantıksal işlemlerin yoğun olduğu durumlarda bu durum daha da önem kazanır. Dışarıdaki bir kaynaktan veri alan (data import) bir veritabanı var ise bu ihtiyaç daha da artar. Yani, veritabanının kullanım yöntemi ve yoğunluğuna göre bir RAM kullanımı gerçekleşir.

SQL Server'ın hızlı ve performanslı I/O işlemleri gerçekleştirilebilmesi için yüksek RAM bir zorunluluktur.

## ÇÖKME SENARYOLARI

Veritabanı uygulamaları, yazılım alanında en kritik alt yapı çalışmalarıdır. İşlem yoğunluğu veritabanı üzerinden gerçekleşen bir uygulamada veritabanının ya da üzerinde bulunduğu sunucunun çökmesi sistemin tamamen kilitlenmesiyle eş degerdir. Kullanılan istemci uygulamada veritabanı yoğunluklu bir işlem olmasa bile, sadece veritabanı üzerinden kullanıcı girişi yapması kritiklik derecesini ortaya koyacaktır. Kullanıcılar uygulamaya giriş yapmak için, yani yazılıma erişebilmek için bile veritabanının çalışıyor ve kullanıcı bilgilerini sorgulayarak yazılıma erişim izni vermesi gerekiyor.

Tüm şirket yazılımlarının tek bir veritabanı kullanan büyük bir şirket yönetim uygulaması olduğunu ve tüm departmanların yetkileri dahilinde bu yazılıma erişerek günlük işlemlerini (veri girişi, raporlama, girdi-çıkı vb.) yaptıklarını düşünelim. Veritabanının çökmesi, yönetim yazılımları çalışmayaçağı için tüm şirket çalışanlarının sorun çözülünceye kadar tatil yapması anlamına gelebilir. Sorun sadece iş yapamamak değildir. Çalışanların saat ücreti üzerinden hesaplama yapıldığında, bu hatanın, maddi açıdan da ne kadar büyük bir maliyeti olacağı hesaplanabilir.

Çökme riskleri özel bir analiz süreci ile hesaplanmalıdır. Veri, kendisini kullanan kullanıcılar ve firma açısından ne kadar önemlidir? Çok kritik seviyede önemlilik arz eden senaryolarda, çeşitli önlemler alınarak çökme sonucu oluşacak veri kaybı ve iş kaybı en aza indirilebilir.

Çökme sonucunda oluşacak sorunları en aza indirmek için aşağıdakiler yapılabilir.

- Veritabanı ve işletim sistemi farklı disk'lerde bulundurulmalı.
- Veritabanı yedekleri belirli aralıklarla otomatik olarak alınmalı.
- Veritabanı yedeğinin işletim sistemi ve veritabanının bulunduğu diskten, hatta mümkünse farklı bir sunucuda bulundurulmalı.
- Veritabanı, dışarıdan erişimlerde belirli IP aralıkları haricinde erişime kapalı olmalı.(Sadece şirket IP'leri gibi)
- Veritabanının bulunduğu sunucu ile istemci arasında yük dengeleme yazılımı kullanılabilir.
- Veritabanının bulunduğu sunucudan farklı bir sunucu ile arasında aynalama yaparak, verinin anlık olarak farklı lokasyonlardaki iki ayrı sunucuya yazılması sağlanabilir.
- **Aynalama:** Birincil olan sunucu üzerinden gerçek işlemler yürütülürken, diğer sunucular, birincil sunucudan anlık işlemlerin kopyasını otomatik olarak alır ve anlık yedek veri olmasını sağlar. Birincil sunucu da çökme yaşandığında, kopya olan ikincil sunucu anında devreye girerek sistemin ayakta kalmasını sağlar.

Daha da ötesinde, veri merkezinde oluşabilecek bağlantı ya da genel saldırı gibi durumlara karşı da veritabanının korumaya alınması gereklidir. Genel olarak

global saldırılarda, veri merkezi hizmetlerini askiya alarak kendini otomatik olarak güvenlik kalkanı içeresine hapseder. Bu durumda verilere ulaşamazsınız. Ancak yasal olarak da yapacağınız bir şey yoktur. Çünkü veri merkezinin genel güvenlik politikaları gereği alt yapısını koruma hakkı vardır. Bu tür durumlarda sorun yaşamamak için, yedek ya da aynalama yapılan sunucuları aynı veri merkezinde barındırmak yerine, farklı lokasyondaki hatta farklı karasal bağlantı hattına sahip bir veri merkezinde tutmak mükemmel en yakın çözüm olabilir.

Çökme senaryolarını hesaplarken asla olasılıklar için imkansız denilmemelidir. Yağmur yağması sonucu, ulusal ve uluslararası hizmet veren bir GSM operatörünün veri merkezine su basması ve büyük veri kaybı yaşanması bir örnek olabilir. Tüm telefon faturalarının 10 günden fazla olan bir kısmının veri kaybı sonucu yok olması nedeniyle, kullanıcılarla hediye edildiğinin söylenmesi (!) gibi bir durumla karşılaşmamak için tüm olasılıkları hesaplamak gereklidir.

Kim bilir, belki veri merkezine meteor düşer!

## **ONLINE SİSTEMLER ÜZERİNDE SQL SERVER**

SQL Server, lisanslı olarak web siteleri ve online veritabanı hizmetlerinde en çok kullanılan veritabanıdır. Bir web uygulaması ya da uzak sunucuya bağlanan veritabanı uygulaması geliştirildiğinde kullanılacak veritabanı çözümüdür.

Bu işlem iyi bir hizmet esnekliği sağlasa da, verinin internet üzerinde olması nedeniyle birçok sorun ve riski de beraberinde getirir.

## **ONLINE GÜVENLİK**

Veritabanının uzak sunucuda olması başlı başına bir güvenlik riskidir. Verinin güvenliği artık sizin elinizde değildir. Uzaktaki sistemin network ve güvenlik yöneticisinin yetenekleri ile sınırlı bir güvenliğe sahipsiniz. Ayrıca verilerinize sizin kadar özen göstermeyeceklerine emin olabilirsiniz. Uzak sunucuya bağlanarak işlem yapabilmeniz için port açılması gereklidir. Port açmak güvenlik açısından risklidir. İyi monitör edilmeli ve yönetilmelidir. Uzaktan bağlanılacak sistemde güvenlik ayarlarının iyi bir network uzmanı tarafından yapılması gereklidir. İyi bir yazılımcı ya da veritabanı uzmanı olmak iyi bir network uzmanı olmak anlamına asla gelmez. SQL Server'in yönetildiği sistem ve ağ doğru yönetilmiyorsa, sizin veritabanı tarafında aldığınız güvenlik önlemlerinin bir önemi yoktur.

## ONLİNE PERFORMANS

İstemciden uzan bir sunucuya bağlanarak veritabanı işlemi yapmaya başlamak, veritabanına ilk bağlanırken bile performansın yavaşlaması anlamına gelir. Veritabanı ve veri yoğunluğuna göre değişecek bu performans sorunu, bir tabloyu **SELECT** ettiğinizde kendisini gösterecektir. Bağlantınız ne kadar yüksek olursa olsun yerel ağ'dan hızlı olamaz. Bu nedenle, geliştirilen veritabanında tüm sorgu ve nesneler daha da küçütlümelii ve gereksiz verilerin filtrelenmesine daha da önem gösterilmelidir. Sorgular optimize edilmeli, gereksiz sunucu bağlantılarından kaçınılmalı, tablo yapıları parçalara bölünerek ağı trafiğini daha az kullanacak şekilde tasarlanmalıdır.

## ONLİNE DESTEK

Uzak sunucu hizmeti alınan firma sizin işinize genelde sizin kadar önem göstermez. Veritabanında ya da sunucu tarafında gerçekleşecek bir hata sonrasında hizmet alınan yerden anında çözüm odaklı destek almak çok önemlidir. Ticari bir çalışma için hazırlanan web sitesinde, kullanıcıların işlem yapması için hazırlanan veritabanında, bir bağlantı hatası oluşması ve bu sorunun çözülmesinin 1 gün sürdüğü senaryosunu düşünmek bile can sıkıcıdır. Daha da ötesinde, uzak sistemlerde veritabanı yedeğini hizmet alınan firma üstlenir. Yedeklerin bulunduğu sunucu ile hizmet verdikleri sunucunun diskinin yandığı gibi en kötü senaryolara hazırlıklı olunmalıdır.

Yasal olarak sorumluluğun karşı tarafta olması, belki paha biçilemez olarak gördüğünüz verilerin geri getirilememesi durumunda bir anlam ifade etmeyecektir. Hizmet alınan firma, veritabanı ve yazılım yedeklerini tutuyor olsa da, verilerin güvenliği açısından kendi yedeklerinizi farklı fiziksel ortamda tutmanız önerilir.

## SORGU OPTİMİZASYONU

Performans işlemlerinin bir diğer unsuru yazılım taraflı performans geliştirmeleridir. Veritabanı tasarımından başlayan yazılım performans unsurları, veritabanı programlama ile devam eder. Donanımsal kısımda çok sık değişiklik gerekmez. Ancak, yazılımsal tarafta hiç bitmeyecek bir performans iyileştirmesi gereklidir. Bir sorgu ilk geliştirildiğinde çok yoğun kullanılmayabilir, fakat kullanıcıların tercihleriyle birlikte sorgu kullanımı artabilir. Bu durumda sorgunun daha hızlı çalışması, sorgu sonucundaki verinin daha hızlı getirilmesi

ve sorgunun ulaştığı tablonun tasarımda geliştirme düzenlemeleri yapma ihtiyacı doğabilir. Bu işlem bir döngüdür ve sürekli devam eder.

## MINIMUM KURALI

Veritabanı ve uygulama için temel kurallardan birisidir. İstemci uygulama tarafından bir veri tablosunun tamamını çekmek çoğu zaman anlamsız ve gereksiz kayıtların sunucudan, network aracılığıyla istemciye ulaşması anlamına gelir. İstemci uygulamanın isteği, tablodaki 10 kaydın 4 sütunu ise, istemciye 20 adet sütun ve içeridiği 100 kaydı iletmek istemci ve sunucu performansı açısından olumsuz bir işlem olacaktır.

Sunucudan istenen her kayıt hem istemicide, hem de sunucuda bellek ve I/O kullanımına sebep olur. Gerekli olan verinin haricinde, fazladan gönderilen her veri bu kaynakları fazladan tüketmek anlamına gelecektir. Performans bakış açısıyla incelenen bir sistemde, istemci yazılım formlarında, veritabanından çekilen ve kullanılmayan her veri performansı israfıdır.

Örneğin, kullanıcının hiç kullanmadığı bir veriyi, veritabanından çekerek uygulamadaki arayüzde görüntülemek bir kaynak israfıdır. Uygulama formları performans işlemleri için tasarlanmalı ve bu bakış açısına göre ilgili veriler listelenmelidir.

Genellikle uygulamalar, sık kullandığı statik veriyi cache bellekte tutarlar. Veritabanında sürekli değişmeyen bir verinin, sürekli veritabanından istenmesi gereksizdir. Örneğin; bir web uygulamasında, ComboBox ile ülkeler listelenirken, bu ülke listesinin her sayfa yenilenmesinde veritabanından çekilmesine gerek yoktur. Uygulama başladığında bir kez veritabanından çekilen ülke listesi, cache belleğe alınır ve sonraki kullanımlarda sürekli cache bellekten statik veri olarak okunur. Cache bellek, doğru kullanıldığında kaynak tüketimini azaltır ve performansa olumlu anlamda katkı sağlar.

## GEÇİCİ TABLOLAR

Genel olarak, hızlı sorgular geliştirmek için kullanılır. Birden fazla tablo birleştirilerek elde edilen bir sorgu senaryosunu ele alalım. Çok sütunlu ve yüz bin kayıt bulunan bir tabloda, sütunlardan birinin tarih bilgisini tuttuğunu düşünelim. Bu tablo üzerinde, Kasım ayına ait bilgiler sürekli olarak sorgulanıyorsa ve bu tarihteki veriler üzerinde sürekli farklı filtre, graplama gibi

işlemler gerçekleştiriliyorsa, Kasım ayına ait verileri geçici bir tabloda tutmak ve sonrasında gerçekleştirilecek tüm sorguları, bu geçici tablo üzerinde yaparak, diğer yüz bin kayıtlı ana veri tablosunu sorgulamadan, daha az veri üzerinde sorgu gerçekleştirmek gerekir.

Bu örnek tam olarak şuna benzer; bir büyük kutu içerisinde tüm eşyaların karışık olarak saklanması ve sonrasında, her seferinde kutu içeriği aranarak bir eşyanın bulunması mı kolay, yoksa büyük kutu içerisinde giysi, mücevher, oyuncak gibi eşyaların ayrılarak küçük kutulara konulması ve her arama isteğinde ilgili kutudan aramak mı daha kolaydır? Tabi ki küçük kutularda, ilgili olduğumuz şeyleri aramak daha kolay olacaktır. Bu işlemde, büyük kutu yüz bin kayıtlık veri tablosu iken, küçük kutu, Kasım ayına ait kayıtlardır.

## FİLTRELEME, GRUPLAMA VE SAYFALAMA

Uygulama tarafından belirtilen bazı özel istekler vardır. Bunlar, veriyi grüplamak, sayfalamak ya da filtrelemek gibi veri üzerinde kaynak ve süre açısından ek yük gerektiren işlemlerdir. Bu tür istekler gerçekleştirildiğinde, öncelikle tam olarak ne istediği belirlenmelidir. Uygulama tarafından gelen istek, maksimum 100 satırdan oluşabilecek bir sorgu içeriyorsa, 100 kaydın üzerinde gönderilen her kayıt ve gereksiz sütun ek kaynak tüketeceği için performansı olumsuz yönde etkileyecektir.

Veri katmanı ile uygulama katmanı arasında iş yükü dağılımının doğru yapılması gereklidir. Veri üzerinde filtreleme, grüplama ya da sayfalaması benzeri işlem gerçekleştirilecekse, bu işlemlerin veritabanında yapılması gereklidir. Örneğin; 1000 satırlık bir veri üzerinde bir sayfalaması işlemi gerçekleştirilecekse, bu kayıt satırlarının tamamını uygulama tarafına göndererek sayfalamanın uygulama katmanında yapılmasını sağlamak network trafiğini ve farklı bazı performans parametrelerini olumsuz etkileyecektir. Sayfalaması için 1000 kayıt isteniyor olabilir. Ancak bu kayıtlar 100 eşit sayfaya bölündüğünde, uygulama katmanında iki ya da üçüncü sayfadaki verinin kullanıcı tarafından talep edileceği belli değildir. Henüz gereksinim duyulmayan bir veri, uygulamaya gönderilmemelidir.

Ayrıca bu tür işlemleri gerçekleştirmek için, indeksler tanımlanmalı ve örneğin, sayfalaması işlemini gerçekleştirecek bir prosedür oluşturularak, kullanıcının, daha önceden sorgu planı hazırlanmış ve derlenmiş olan bu prosedüre,

sayfa bilgilerini ileterek, belirlediği sayfa aralıklarındaki bilgileri elde etmesi sağlanabilir.

Filtreleme işlemlerinde, uygulama katmanına,其实需要的用户来说，它可能是一个很大的负担。例如，如果一个用户想要找到所有姓“A”且名字以“i”开头的记录，那么系统需要遍历所有的记录并进行比较。这在大数据量的情况下可能会非常耗时。因此，过滤操作通常会增加查询的成本。

Gruplama işlemlerinde, uygulama katmanı genel olarak, belirli ve az sayıda sonuç bekler. Gruplama isteği tam olarak bilinmeli ve bu isteğin karşılanması şeklinde gruplama sorgusu hazırlanıktan sonra, uygulama tarafına iletilmelidir. Diğer sorgu yöntemlerinde olduğu gibi, gruplama işlemlerinde de yapılan istekle ilgili olmayan hiç bir kayıt uygulama katmanına iletilememelidir.

## İNDEKS

Veritabanı yönetimi ve performanslı çalışması için en çok ihtiyaç duyulan veritabanı nesnesidir. SQL Server, sorguların daha performanslı çalışabilmesi için indeksleri kullanır. SQL Server, tüm sorgularda kullanmak için arka planda bazı özel tanımlamalar ve veri tanıma parametreleri kullanır. Çeşitli veritabanı istatistikleriyle birlikte hangi tablo, sütun ve sorguların daha çok kullanıldığını öğrenerek kendisi performans artırıcı önlemler alabilir. Bir T-SQL geliştiricisi, veritabanı sorgu performansını artırmak için çok kullanılan ve filtrelemeler gerçekleştirilen kayıtlar için indeks tanımlayabilir. İndeks kullanımı performans artırmak için kullanılsa da, doğru tasarılanmayan indeks mimarisi, verinin olduğundan daha yavaş çalışmasına da sebep olabilir. Sorgularda kullanılan **WHERE** ile filtreleme, **JOIN** ile birleştirme işlemleri, **ORDER BY** ile sıralama işlemleri için indeksler tanımlanmalıdır.

İndeksler ile ilgili aşağıdaki öneriler dikkate alınmalıdır.

- Tabloların veritabanı sunucusu tarafından doğru taranabilmesi için, her tabloda bir **Primary Key** bulunmalı.
- Sık kullanılan **WHERE** filtrelemelerinde kullanılan alanlarda **NonClustered** indeks kullanılmalı.

- Sık kullanılan ve genellikle beraber `WHERE` ile filtrelenen sütunlar, birlikte indekslenmeli.
- `JOIN` ile birleştirilen tablolarda `Foreign Key` alanlarının `NonClustered` olarak indekslenmeli.

İndekslerin nerede, ne zaman kullanılması gerektiğini ve performans etkilerini görmek için **çalıştırma planı** (*execution plan*) incelenebilir. Ayrıca SQL Server, Index Tuning Wizard ile sizin için bazı indeks önerileri sunar.

## FİLTRELİ İNDEKS (FILTERED INDEX)

SQL Server 2008 ile birlikte gelen, optimize edilmiş bir `NonClustered` indeks'tir. Filtreli indeks oluşturulurken kullanılan `WHERE` filtresi nedeniyle indeks key'i, tüm verileri değil, sadece belirlenen filtreye uygun veriyi içerir. Sorgular genel olarak `WHERE` ya da benzeri koşullar ilefiltrelerek elde edilir. Bu nedenle, filtrelili sorgular üzerinde indeks oluşturmak daha performanslı olabilir. Filtreli indeksler bakım maliyetini düşürecegi gibi, disk üzerinde de daha az yer kaplar.

### SORGU PERFORMANSINI ARTIRIR

Filtreli indeksler daha az veri içerdigi için indeks üzerinden yapılacak tarama işlemleri normal indekslere göre daha hızlı yapılacaktır. Ayrıca bu işlem, verinin veritabanına daha doğru işlemesi için tanıtılması anlamına gelir. Bu durumda veritabanı, oluşturacağı istatistik ile veriyi daha iyi işleyecek parametrelere sahip olacaktır.

### BAKIM MALİYETLERİNİ DÜŞÜRÜR

Tablo üzerinde DML işlemleri arttıkça, indeks **fragmante** (*parçalanma*) olur. Filtreli indeks, daha az veri içerdigi için fragmante sorunu olasılığı daha da düşük olacaktır. Ayrıca, fragmante olsa bile, bakım işlemi filtrelili indekslerde daha az veri üzerinde yapılacağı için, bakım maliyeti daha az olur.

### DİSK DEPOLAMA MALİYETİNİ DÜŞÜRÜR

Normal indekslere göre filtrelili indeksler, daha az veri içerdigi için, kapladığı disk alanı da düşük olacaktır. Bu durum, sadece disk kotası ile ilgili değil, I/O, CPU ve RAM üzerinde de maliyetleri azaltmak için faydalıdır.

## VIEW

Veritabanında en çok kullanılan sorgu komutu, veri seçme için kullanılan **SELECT** komutudur. Hem en çok hem de en karmaşık sorgularda bu komut kullanılır. Bu nedenle SQL Server, veri seçme işlemlerini daha performanslı hale getirebilmek için birçok ek özellik sunar. **View (görünüm)** nesneleri, filtreli,filtresiz ya da **JOIN** ile birleştirilmiş bir veri kümesine takma isim vererek, gerçek bir tablo gibi sorgulanmasını sağlar. Kitabın view bölümünde derinlemesine incelediğimiz bu konu için, performanslı sorgular hazırlanması gereklidir.

Bir tabloda tüm veriler seçilecek ise, view kullanılmasının amacı nedir? View kullanmak için gerçek ihtiyaçlara sahip olunmalıdır. Örneğin; bir **JOIN** işlemi sonucunda oluşturulacak verinin, tekrar tekrar yazılmaması için view ile isimlendirerek hızlı erişmek doğru bir tercih olabilir. Ancak herhangi bir filtreleme yapmadan, sadece bir tablonun 3 sütununun değerini verecek bir view oluşturmak anlamlı ve doğru değildir. Çünkü oluşturulan her view, aslında veri ile uygulama arasına yerleştirilen ara bir katmandır. Ara katmanlar her zaman ek iş yükü getirir ve işlemin az ya da çok, yavaşmasına sebep olur.

View kullanımının performans üzerindeki etkisini inceledik. Bazı durumlarda view kullanmanın performansı artıracağı senaryolarda gerçekleşebilir. Bu tür durumlarda, view içerisindeki sorguyu akıllıca hazırlamak gereklidir. Örneğin; bir view içerisinde sıralama işlemi yapmak anlamsızdır. Çünkü view bir tablo gibi sorgulanabilir. Geliştirici, view içerisinde sıralamayı A şeklinde yaparsa, view'i kullanan kişi sıralamayı B'ye göre yapmak isteyebilir. Bu durumda, view içerisinde yapılan sıralama işlemi, kaynak ve işlem süresini gereksiz şekilde kullanıyor olacaktır. Aynı zamanda, view'in kullanılacağı yerde yapılması gereken dönüştürme ya da hesaplama gibi işlemlerin de view içerisinde yapılmaması gereklidir.

## CONSTRAINT

Constraint'ler, veritabanı için değer nesneleridir. Ancak her şeyin bir bedeli vardır. Veri girişi sırasında kullanılan bir Check Constraint'ın görevi verinin, istenen şartı uygunluğunu kontrol etmektir. Bu da veri girişinde yavaşlamaya neden olacaktır. Veri girişinde, zorunlu kalınmadığı taktirde, bu tür kontrollerden kaçınılmalıdır. Veritabanı katmanında yapılacak kontrolleri, uygulama katmanında yapmak bir çözüm olabilir. Örneğin; bir kullanıcı

tablosunda email bilgisinin benzersiz olması gereklidir. Email bilgisi kontrolden geçirilmelidir. Bu kontrol, veritabanı katmanında Check Constraint ile yapılabileceği gibi, uygulama katmanında birkaç satır kod ile de yapılabilir. Veri ekleme performansı düşünülmüşse, bu işlemin uygulama katmanında yapılması gereklidir.

## TRIGGER

Trigger'lar etkili ve ileri seviye bir veritabanı özelliğidir. Trigger'lar otomatik olarak tetiklendiği için, bazı sorunları da beraberinde getirebilir. Trigger kullanımı konusunda dikkatli davranış gereklidir. Her veritabanı nesnesinde olduğu gibi, trigger'lar için de performans artırıcı ve doğru kullanmayı sağlayacak bazı püf nokta ve öneriler vardır. Bu bölümde, trigger tabanlı performans konularına değineceğiz.

### TRIGGER'LAR REAKTİF'TİR

Proaktif, bir olaydan önce gerçekleşmeyi gerektirirken, reaktif bir olaydan sonra gerçekleşmesi anlamına gelir. Trigger başlatıldıkten sonra tüm sorgu çalışır ve transaction log bilgisi tutulur. Ancak, trigger'larda da her şey yolunda gitmeyecektir. Trigger işlemini gerçekleştiren transaction'da bir hata meydana geldiğinde, tüm transaction işlemleri ROLLBACK ile geri alınır. Trigger performansı ile trigger gövdesinde tanımlanan SQL kod bloğunun büyüklüğü doğru orantılıdır. Trigger ne kadar çok iş yapıyorsa, hata sonucundaki ROLLBACK işlemi o kadar uzun sürecek ve performansı olumsuz yönde etkileyecektir.

### TRIGGER'LARDA ORTAK ZAMANLILIK

Trigger ile trigger'ı başlatan ifade dolaylı olarak transaction'ın birer parçasıdır. Trigger'ı başlatan ifade, açık bir transaction olmasa bile, SQL Server için tek ifadeli bir transaction'dır. Her ikisini de bir bütün olarak işler. Bir transaction parçaları olduğu için ROLLBACK ile işlem geri alınması durumunda tüm yapılan işlemler geri alınır. Trigger bölümünde seviye kavramından bahsettiğimiz gibi, Trigger'lar 32 seviyeye kadar derinliği desteklemektedir. Her trigger ve bu trigger'ların tetiklenmesini sağlayan SQL işlemlerinin kod ve işlem yoğunluğuna göre, ROLLBACK işleminin ne kadar zor olacağı hesaplanabilir.

## TRIGGER GENİŞLİĞİ

Trigger genişliği kavramı genel olarak tüm SQL sorgularını kapsayan bir performans sorunudur. Bir kod bloğunun çalışma süresi ve performansı içerisinde barındırdığı işlemlerin ve kodların yoğunluğu ile ilgilidir. Kendisini tetikleyen bir transaction'ın parçası olarak çalışan trigger'lar için sorgu hızı ve kod yoğunluğu önemli bir etkendir.

Bir Stored Procedure tarafından tetiklenen trigger tamamlanmadan, Stored Procedure içerisindeki transaction'da tamamlanmamış olacaktır. Bu durumda performansı sadece trigger açısından değil, trigger'i tetikleyen transaction ile birlikte düşünmek gereklidir. Bir trigger tetikleyen Stored Procedure'un performans analizleri yapılmırken, düşük performans ile çalışması durumunda, yapılan prosedür incelemelerinde kod bloğunun performansı yüksek olabilir. Ancak, tetiklediği trigger'in performansı düşük olduğu için bu durum prosedürün performansını da olumsuz yönde etkileyecektir.

Trigger kullanırken, tetikleyen transaction ile trigger gövdesini mantıksal olarak birlikte ele almak gereklidir.

## TRIGGER'LAR VE ROLLBACK

Trigger'ları anlatırken, özellikle geri alma (**ROLLBACK**) işleminin önemine deðindik. **AFTER** trigger gibi önce işlemi yaparak, sonrasında trigger işlemlerini gerçekleştiren trigger'larda bir hata olusma durumunda ne kadar işlemin geri alınacağı iyi hesaplanmalıdır. Bu sorunun önüne geçmek T-SQL geliştiricinin görevidir. Trigger'i tetikleyen transaction'ın ve trigger içerisindeki işlemlerin hata olasılıklarını önceden hesaplamalı ve bu olası sorumlara karşı önlem alınmalıdır.

Trigger performansı açısından en önemli unsurlardan biridir. Genel kavram olarak, trigger içerisinde **ROLLBACK** işlemine gerek bırakmadan geliştirme yapmak gerektiği söylenebilir.

## PERFORMANS İÇİN İSTATİSTİKSEL VERİ KULLANIMI

Bir iş için performans artırımı gerçekleştirmek için, o işin veri parametreleri hakkında bilgi sahibi olunmalıdır. İstatistiksel bilgilere sahip olunmadan

performans işlemleri gerçekleştirilemez. Kısmen gerçekleştirilse bile, gerçekten performans sağlanıp sağlanamadığı konusunda gerekli bilgiler elde edilemez. Veritabanında performans analizi yapabilmek için gerekli tüm istatistikler SQL Server tarafından tutulurlar. Bu istatistikleri kullanarak, hangi sorgunun nasıl çalıştığı, sorgu planı, sorgu süresi, kaynak tüketimi, I/O ve CPU kullanımı, kullanılan bellek miktarı ve sistemde ayrılan bellek, disk, CPU miktarı gibi bilgilere erişilebilir. Sistem kaynakları hakkında bilgi sahibi olunmadan bir sistemin performansını analiz etmek mümkün değildir.

Bu bölümde, SQL Server'da veri ve donanım alt yapısı ile ilgili istatistiksel verileri incelemeyi, kullanmayı ve analiz etme konularını işleyeceğiz.

## **PERFORMANS İÇİN DMV VE DMF KULLANIMI**

SQL Server'da veri yönetimi ve verinin sistem tarafından bilgi ve istatistiklerini elde edebilmek için geliştirilen view (DMV) ve fonksiyonlardır (DMF). DMV ve DMF'ler disk üzerinde değil, bellek üzerinde kümülatif olarak veri tutarlar. Bu nedenle bu tür sorgular ile elde edilen veriler, SQL Server servisi başlatıldıkten sonra toplanan veriler olduğu için, istatistik açısından kesin kanaate varılamayabilir. DMV ve DMF ile alınan verilerin istatistiksel olarak kritik derecede kullanılabilmesi için SQL Server servisinin uzun süredir çalışıyor olması gereklidir.

Açılımları aşağıdaki gibidir:

- DMV (*Dynamic Management View*)
- DMF (*Dynamic Management Function*)

Bu bölümde, DMV ve DMF kullanarak, performans için gerekli bilgileri elde etmeye çalışacağız. Bölümde DMF ve DMF'nin derinliklerine inmemekle birlikte, kritik konulardaki DBA ve geliştiriciye sağlayacağı önemli bilgileri elde edeceğiz.

Bu bölümle birlikte, bir başlangıç yapabileceğiniz DMV ve DMF'ler ile indeksler hakkında bilgiler alarak, kullanılan ve kullanılmayan ya da eksik indeksleri tespit etmek gibi performans açısından gerekli bilgiler elde edebilirsiniz.

## DMV VE DMF HANGİ AMAÇ İLE KULLANILIR?

DMV ve DMF ile sistem tarafındanki problemleri belirleme, performans ayarlarının gözlenmesi ve sistemin izlenmesi gibi istatistiksel veri takibi gerçekleştirilebilir.

- Performans:** Sistemdeki bozuk indeks, eksik indeks ve yüksek I/O işlemi gerçekleştiren sorgular belirlenerek bunların düzenlenmesi için kullanılabilir.
- Problemleri Gidermek:** Sistem üzerinde bekleyen ya da çalışması engellenen işlemlerin belirlenmesi için kullanılabilir.
- Sistemi İzlemek:** Sistemde hangi kullanıcının ne tür işlem yaptığı belirlemek ya da execution planlarının incelenmesi gibi işlemler için kullanılabilir.

DMV ve DMF'ler yapısal olarak basit olmakla birlikte, `JOIN`'li kullanımı ve elde edilen verilerin sistem taraflı teknik bilgiler içermesi nedeniyle karmaşık gelebilir. Bu yöntemler ile elde edilen verileri zaman içerisinde tecrübe edinerek anlayabilir ve ihtiyaçlarınız doğrultusunda farklı kombinasyonlar oluşturarak geliştirebilirsiniz.

Şimdi, temel olarak bazı örnekler gerçekleştirerek DMV ve DMF'lerin yeteneklerini inceleyelim.

## BAĞLANTI HAKKINDA BİLGİ ALMAK

SQL Server'a gerçekleştirilen bağlantı ile ilgili bilgileri elde etmek için kullanılır.

---

```

SELECT C.session_id,
       C.auth_scheme,
       C.last_read,
       C.last_write,
       C.client_net_address,
       C.local_tcp_port,
       ST.text AS lastQuery
FROM sys.dm_exec_connections C
CROSS APPLY sys.dm_exec_sql_text(C.most_recent_sql_handle) ST
  
```

---

	session_id	auth_scheme	last_read	last_write	client_net_address	local_tcp_port	lastQuery
1	51	NTLM	2013-02-15 15:29:03.460	2013-02-15 15:29:03.463	<local machine>	NULL	CREATE PROCEDURE [dbo].[CleanEventRecords] @M...
2	52	NTLM	2013-02-15 15:30:38.637	2013-02-15 15:30:38.637	<local machine>	NULL	declare @BatchID uniqueidentifier ...
3	53	NTLM	2013-02-15 15:30:21.240	2013-02-15 15:30:21.240	<local machine>	NULL	select value_in_use from sys.configurations where configur...
4	54	NTLM	2013-02-15 15:30:40.270	2013-02-15 15:30:40.270	<local machine>	NULL	SELECT C.session_id, C.auth_scheme, C.last_read, C.l...

- **session\_id**: Bağlantının kullandığı SessionID değerini verir.
- **auth\_scheme**: Bağlantının SQL Server’mı, Windows Authentication’mı olduğu bilgisini verir.
- **last\_read**: Bağlantının en son okuma yaptığı zaman.
- **last\_write**: Bağlantının en son yazma yaptığı zaman.
- **client\_net\_address**: Bağlantıyı yapan makinenin IP adresi.
- **local\_tcp\_port**: Bağlantının hangi port üzerinden yapıldığı bilgisi.
- **lastQuery**: Bağlantının üzerinden çalıştırılan en son sorgu.

## SESSION HAKKINDA BİLGİ ALMAK

Session hakkında bilgi almak için `sys.dm_exec_sessions` isimli DMV kullanılabilir. Sisteme ne zaman login olunduğu, session’ı açan program ve makinenin bilgisi gibi bir çok bilgi sunar.

---

```
SELECT login_name, COUNT(session_id) AS session_count, login_time
FROM sys.dm_exec_sessions
GROUP BY login_name, login_time;
```

---

	login_name	session_count	login_time
1	sa	1	2013-02-15 10:43:09.967
2	sa	7	2013-02-15 10:43:09.980
3	sa	1	2013-02-15 10:43:11.767
4	sa	1	2013-02-15 10:43:12.250
5	sa	2	2013-02-15 10:43:12.297
6	sa	2	2013-02-15 10:43:40.847
7	sa	1	2013-02-15 10:43:49.000

## VERİTABANI SUNUCUSU HAKKINDA BİLGİ ALMAK

CPU, I/O ve RAM ile ilgili kararlar vermek ve performans analizi yapabilmek için veritabanı sunucusu hakkında bazı bilgilere sahip olmak gerekebilir. Veritabanı sunucusunun ne zaman başlatıldığı, en son ne zaman başlatıldığı, CPU adeti ve fiziksel hafıza gibi daha bir çok sunucu ile ilgili temel seviye bilgileri elde etmek için aşağıdaki DMV kullanılır.

---

```
SELECT * FROM sys.dm_os_sys_info;
```

---

Sorgu sonucunda listelenen sütunları inceleyelim.

- **sqlserver\_start\_time**: SQL Server servisinin ne zaman başlatıldığı.
- **ms\_ticks**: Veritabanı sunucusunun en son başlatıldığı zamandan beri geçen süre(milisaniye).
- **sqlserver\_start\_time\_ms\_ticks**: SQL Server servisi başladığında ms\_ticks süresi.
- **cpu\_count**: Sunucunun sahip olduğu CPU sayısı.
- **physical\_memory\_kb**: Sunucunun fiziksel hafıza boyutu.
- **virtual\_memory\_kb**: Sunucunun sanal hafıza boyutu.
- **max\_workers\_count**: Maksimum kaç thread'in oluşturulabileceği bilgisi.
- **scheduler\_count**: Kullanıcı işlerini yapacak scheduler bilgisi.
- **scheduler\_total\_count**: Toplam scheduler adeti.

Performans analizini gerçekleştirirken SQL Server'ın ne kadar süredir çalıştığını öğrenmek gerekebilir. Şimdi, SQL Server'ın kaç dakikadır çalıştığını bulalım.

---

SELECT

DATEDIFF(MINUTE, sqlserver\_start\_time, CURRENT\_TIMESTAMP) AS UpTime  
FROM sys.dm\_os\_sys\_info;

---

Aynı işlemi gerçekleştirecek bir başka yöntem de, **ms\_ticks** ve **sqlserver\_start\_time\_ms\_ticks** sütunlarını kullanmaktadır.

	UpTime
1	289

---

SELECT (ms\_ticks-sqlserver\_start\_time\_ms\_ticks)/1000/60 AS UpTime  
FROM sys.dm\_os\_sys\_info;

---

## FİZİKSEL BELLEK HAKKINDA BİLGİ ALMAK

Daha önce sunucu ve dosyalar ile ilgili DMV kullanımlarını inceledik. SQL Server kullanımında performansı etkileyen en önemli unsurlardan biri de bellek kullanımıdır. Fiziksel bellek ile ilgili istatistiksel bilgileri elde etmek için **sys.dm\_os\_sys\_memory** isimli DMV kullanılır.

---

SELECT \* FROM sys.dm\_os\_sys\_memory;

---

- **total\_physical\_memory\_kb**: Sunucuda bulunan toplam fiziksel bellek değeri.
- **available\_physical\_memory\_kb**: Sunucuda bulunan fiziksel belleğin ne kadarının boşta olduğu bilgisi.
- **totalpage\_file\_kb**: Page File'ın toplam ulaşabilecegi boyut.
- **available\_page\_file\_kb**: Kullanılabilen Page File boyutu.
- **system\_memory\_state\_desc**: Bellek durumuna göre, aşağıdaki açıklama mesajlarından birini gösterir.
  - Available physical memory is high
  - Available physical memory is low
  - Physical memory usage is steady

Her sistemde değişken değerlere sahip olacak bellek değerlerini almak için gerekli DMV sorgusunu hazırlayalım.

---

```
SELECT
    total_physical_memory_kb/1024 AS [ToplamFizikselBellek],
    (total_physical_memory_kb-available_physical_memory_kb)/1024
        AS [KullanılanFizikselBellek],
    available_physical_memory_kb/1024 AS [KullanılabilirFizikselBellek],
    total_page_file_kb/1024 AS [ToplamPageF(MB)],
    (total_page_file_kb - available_page_file_kb)/1024 AS [KullanılanPageF(MB)],
    available_page_file_kb/1024 AS [KullanılabilirPageF(MB)],
    system_memory_state_desc
FROM
    sys.dm_os_sys_memory;
```

---

	ToplamFizikselBellek	KullanılanFizikselBellek	KullanılabilirFizikselBellek	ToplamPageF(MB)	KullanılanPageF(MB)	KullanılabilirPageF(MB)	system_memory_state_desc
1	8172	3359	4812	16343	4801	11541	Available physical memory is high

## AKTİF OLARAK ÇALIŞAN İSTEKLERİ SORGULAMAK

SQL Server üzerinde, şuan aktif olan istekleri sorgulamak için `sys.dm_exec_requests` isimli DMV kullanılır. Bu isteklerin durumunu, sorgu metnini ve hangi sorguda beklediği gibi birçok bilgi elde edilebilir.

- **dbname**: Session'ın çalıştığı veritabanının ismi.
- **login\_name**: Session'ın hangi kullanıcı adı ile açıldığı bilgisi.

- **host\_name**: Session'ın makine bilgisi.
- **text**: Çalışmaya devam eden sorgunun metnini gösterir.
- **statement\_text**: Çalışmaya devam eden sorgunun şuan çalışmakta olan kısmını gösterir. 100 satırlık bir sorgu çalışıyorsa, çalışma sırası 45. satırda ise, buradaki sorguları gösterecektir.
- **blocking\_session\_id**: Başka bir session tarafından bir session bloklandıysa, bu session'ı bloklayan session'ın ID değeri gelir.
- **status**: Sorgunun şu anki durumunu gösterir.
- **wait\_type**: Sorgu şu an bloklu durumdaysa, bu blok'un tipini verir.
- **wait\_time**: Sorgu şu an bloklu durumdaysa, ne kadar zamandır (*milisaniye*) bloklu olduğunu verir.
- **percent\_complete**: İşlem gerçekleştirken, işlemin yüzde kaçında olduğunu verir.
- **estimated\_completion\_time**: İşlem gerçekleştirken, işlemin tamamlanmasına kaç milisaniye kaldığını verir.

Bir sorgu ekranı açarak aşağıdaki döngü oluşturacak sorguyu çalıştırıyalım.

---

```
DECLARE @test INT = 0;
WHILE 1 = 1
SET @test = 1;
```

---

Döngü devam ederken, yeni bir sorgu ekranı açarak aşağıdaki sorguyu çalıştırıyalım.

---

```
SELECT DB_NAME(er.database_id) AS DB_ismi,
es.login_name,
es.host_name,
st.text,
SUBSTRING(st.text, (er.statement_start_offset/2) + 1,
((CASE er.statement_end_offset
WHEN -1 THEN DATALENGTH(st.text)
ELSE er.statement_end_offset
END - er.statement_start_offset)/2) + 1) AS ifade_metni,
er.blocking_session_id,
er.status,
```

```

        er.wait_type,
        er.wait_time,
        er.percent_complete,
        er.estimated_completion_time
    FROM sys.dm_exec_requests er
        LEFT JOIN sys.dm_exec_sessions es ON es.session_id = er.session_id
        CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) st
        CROSS APPLY sys.dm_exec_query_plan(er.plan_handle) qp
    WHERE er.session_id > 50 AND er.session_id != @@SPID;

```

---

DB_ismi	login_name	host_name	text	fade_metni	blocking_session_id	status	wait_type	wait_time	percent_complete	estimated_completion_time
1 AdventureWorks2012	djbil\pc\djbil	DUBIL-PC	DECLARE @test INT = 0; WHILE 1 = 1 SET @test =... WHILE 1 = 1	0	running	NULL	0	0	0	0

## CACHE'LENEN SORGU PLANLARI HAKKINDA BİLGİ ALMAK

SQL Server, çalıştırılan tüm SQL sorguları (adhoc, prosedür, trigger, view vb.) için hazırlanan sorgu planlarını saklar. Aynı sorgunun bir sonraki çalıştırılmasında, saklanan bu sorgu planlarını kullanarak performans elde eder.

Cache'lenen sorgu planları hakkında bilgi almak için `sys.dm_exec_cached_plans` isimli DMV kullanılır.

---

```
SELECT * FROM sys.dm_exec_cached_plans;
```

---

Sorgu sonucunda listelenen sütunları inceleyelim.

- **usecounts**: Cache'lenen sorgu planının, cache'lendikten sonra kaç defa kullanıldığı gösterir.
- **size\_in\_bytes**: Cache'lenen sorgu planının hafızada kaç bayt yer kapladığını gösterir.
- **objtype**: Cache'lenen nesnenin tipini (Adhoc, Proc, Trigger vb.) verir.
- **plan\_handle**: Cache'lenen nesnenin planını verir.

Cache'lenmiş nesnelerden, cache'lendikten sonra 3 den fazla kullanılanları listeleyelim.

---

```

SELECT CP.usecounts,
ST.text,
QP.query_plan,
CP.cacheobjtype,
CP.objtype,
CP.size_in_bytes
FROM sys.dm_exec_cached_plans CP
CROSS APPLY sys.dm_exec_query_plan(CP.plan_handle) QP
CROSS APPLY sys.dm_exec_sql_text(CP.plan_handle) ST
WHERE CP.usecounts > 3
ORDER BY CP.usecounts DESC;

```

---

	usecounts	text	query_plan	cacheobjtype	objtype	size_in_bytes
1	1737	declare @BatchID uniqueide...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	Compiled Plan	Adhoc	229376
2	1737	declare @BatchID uniqu...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	Compiled Plan	Adhoc	147456
3	841	0 select table_id, item_guid, opsln_fseqno, opsln_bOf...	NULL	Compiled Plan	Prepared	32768
4	841	0 select table_id, item_guid, opsln_fseqno, opsln_bOf...	NULL	Compiled Plan	Prepared	32768
5	841	0 select table_id, item_guid, opsln_fseqno, opsln_bOf...	NULL	Compiled Plan	Prepared	32768
6	841	0 select table_id, item_guid, opsln_fseqno, opsln_bOf...	NULL	Compiled Plan	Prepared	32768
7	841	0 select table_id, item_guid, opsln_fseqno, opsln_bOf...	NULL	Compiled Plan	Prepared	32768

Stored Procedure kullanırken, sorgu performansını hesaplamak için sorgu planlarını incelemek ve gerekli T-SQL müdahalelerini prosedür içerisinde gerçekleştirmek gerekir. Şimdi, cache'lenen Stored Procedure'lerin sorgu planlarını elde edelim.

---

```

SELECT
DB_NAME(st.dbid) AS DB_ismi,
OBJECT_SCHEMA_NAME(ST.objectid, ST.dbid) AS sema_ismi,
OBJECT_NAME(ST.objectid, ST.dbid) AS nesne_ismi,
ST.text,
QP.query_plan,
CP.usecounts,
CP.size_in_bytes
FROM sys.dm_exec_cached_plans CP
CROSS APPLY sys.dm_exec_query_plan(CP.plan_handle) QP
CROSS APPLY sys.dm_exec_sql_text(CP.plan_handle) ST
WHERE ST.dbid <> 32767
AND CP.objtype = 'Proc';

```

---

DB_İsmi	sema_İsmi	nesne_İsmi	text	query_plan	usecounts	size_in_bytes
1 msdb	dbo	fn_syspolicy_is_automation_enabled	CREATE FUNCTION fn_syspolicy_is_automation_enabled() ...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	1	147456
2 Report Server	dbo	CleanExpiredEditSessions	CREATE PROC [dbo].[CleanExpiredEditSessions] @Max...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	524288
3 Report Server	dbo	CleanExpiredJobs	CREATE PROCEDURE [dbo].[CleanExpiredJobs] AS SET...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	57344
4 Report Server	dbo	CleanOrphanedSnapshots	CREATE PROCEDURE [dbo].[CleanOrphanedSnapshots] ...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	1556480
5 Report Server	dbo	CleanExpiredCache	CREATE PROCEDURE [dbo].[CleanExpiredCache] AS SE...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	147456
6 Report Server	dbo	DeleteExpiredPersistedStreams	CREATE PROCEDURE [dbo].[DeleteExpiredPersistedStrea...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	73728
7 Report Server	dbo	CleanExpiredSessions	CREATE PROCEDURE [dbo].[CleanExpiredSessions] @Se...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...>	29	532480

Yukarıdaki sorguyu çalıştırıldıktan sonra listelenen sonucu inceleyin. Daha sonra, aşağıdaki gibi iki prosedür çalıştırarak tekrar sorgulayın. Kendi çalıştırığınız prosedürün sorgu planını sonuçlar içerisinde göreceksiniz.

## CACHE'LENEN SORGU PLANLARIN NESNE TİPİNE GÖRE DAĞILIMI

Cache'lenen sorguların hangi nesne tipinde oldukları SQL Server'da kayıtlıydı. Bu nesne tipi değerlerini kullanarak sorgu planlarının nesne tipine göre gruplanmasını ve hangi nesne tipinde kaç adet sorgu planı olduğu hesaplanabilir.

---

```
SELECT CP.objtype,COUNT(*) AS nesne_toplam
FROM sys.dm_exec_cached_plans CP
GROUP BY CP.objtype
ORDER BY 2 DESC;
```

---

objtype	nesne_toplam
1 View	90
2 Adhoc	58
3 Prepared	41
4 Proc	19
5 Check	6
6 UsrTab	1

## PARAMETRE OLARAK VERİLEN SQL\_HANDLE'IN SQL SORGUSUNU ELDE ETMEK

`sys.dm_exec_sql_text` isimli DMF'ye, parametre olarak verilen `sql_handle`'nın, SQL sorgu metnine ulaşılabilir. Örneğin; SQL Server'da aktivitelerin takip edilmesini sağlayan **Activity Monitor**'de bir **Session ID** belirlenerek, bu process'in SQL sorgu metnine ulaşılabilir.

---

```
SELECT ST.text
FROM sys.dm_exec_requests R
```

```
CROSS APPLY sys.dm_exec_sql_text(sql_handle) ST
WHERE session_id = @@SPID;
```

text

```
1 SELECT ST.text FROM sys.dm_exec_requests R CROSS APPLY sys.dm_exec_sql_text(sql_handle) ST WHERE session_id = @...
```

Aktif olan bağlantıların uyguladıkları son SQL script'leri de öğrenilebilir.

```
SELECT ST.text
FROM sys.dm_exec_connections C
CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) ST
WHERE session_id = @@SPID;
```

text

```
1 SELECT ST.text FROM sys.dm_exec_connections C CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) ST ...
```

**sql\_handle** bilgisine erişmek ya da **CROSS APPLY** ile birbirine bağlamak için aşağıdaki DMV'ler kullanılabilir.

```
sys.dm_exec_sql_text
sys.dm_exec_requests
sys.dm_exec_cursors
sys.dm_exec_xml_handles
sys.dm_exec_query_memory_grants
sys.dm_exec_connections
```

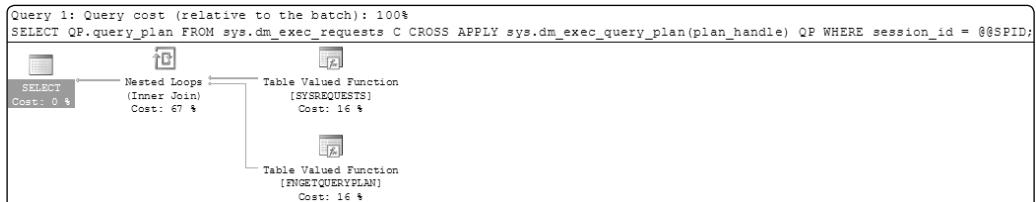
## PARAMETRE OLARAK VERİLEN PLAN\_HANDLE'IN SORGU PLANINI ELDE ETMEK

**plan\_handle** olarak verilen sorgunun XML formatında sorgu planını elde etmek için kullanılır.

Aktif olarak devam eden process'lerin sorgu planlarını elde edelim.

```
SELECT QP.query_plan
FROM sys.dm_exec_requests C
CROSS APPLY sys.dm_exec_query_plan(plan_handle) QP
WHERE session_id = @@SPID;
```

Sorgu sonucunda gelen XML bağlantı açıldığında sorgu planı, aşağıdaki gibi görsel olarak görüntülenecektir.



## SORGU İSTATİSTİKLERİ

`sys.dm_exec_query_stats` isimli DMV kullanılarak sorgu istatistikleri elde edilebilir. Bu DMV ile, cache'lenmiş sorguların CPU süreleri (*worker time*), toplam çalışma süreleri (*elapsed time*), fiziksel okuma (physical read), mantıksal okuma (logical read), mantıksal yazmalar (logical write) gibi bir çok parametre analiz edilerek sorunlu sorgular üzerinde iyileştirme çalışmaları yapabilmek için SQL Server yönetici ve geliştiricilerine istatistiksel bilgiler verir.

---

```
select * from sys.dm_exec_query_stats;
```

---

Sorgu sonucunda 5 ana bilgi gelmektedir. Bunlar;

- **workertime:** Sorgunun CPU üzerinde geçirdiği süre.
- **elapsedTime:** Sorgunun toplam süresi.
- **physicalread:** Diskten yapılan okuma miktarı.
- **logicalread:** Bellekten yapılan okuma miktarı.
- **logicalwrite:** Belleğe yapılan yazma miktarı.

Bu istatistik özelliklerinin her biri için 4 farklı istatistik sütunu yer alır. Örneğin; workertime özelliği için, `execution_count`'un 10 olduğunu düşünürsek;

- **min\_worker\_time:** Sorgunun 10 kez çalışmasında, CPU üzerinde minimum zamanı geçirenin kaç microsaniye olduğunu gösterir.
- **max\_worker\_time:** Sorgunun 10 kez çalışmasında, CPU üzerinde maximum zamanı geçirenin kaç microsaniye olduğunu gösterir.
- **last\_worker\_time:** `Last_Execution_Time` olarak verilen zamanda, çalışan sorgunun CPU üzerinde kaç microsaniye iş yaptığı gösterir.

- total\_worker\_time:** Sorgunun 10 kez çalışmada toplamda kaç microsaniye CPU üzerinde iş yaptığıını gösterir.

Şimdi birkaç örnek yaparak sorgu üzerinde deneyelim.

En fazla CPU tüketen ilk 20 sorguyu listeleyelim.

---

```

SELECT
    q.[text],
    SUBSTRING(q.text, (qs.statement_start_offset/2)+1,
        ((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(q.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2) + 1) AS ifade_metni,
    qs.last_execution_time,
    qs.execution_count,
    qs.total_worker_time / 1000000 AS toplam_cpu_zaman_sn,
    qs.total_worker_time / qs.execution_count / 1000 AS avg_cpu_zaman_ms,
    qp.query_plan,
    DB_NAME(q.dbid) AS veritabani_ismi,
    q.objectid,
    q.number,
    q.encrypted
FROM
    (SELECT TOP 20
        qs.last_execution_time,
        qs.execution_count,
        qs.plan_handle,
        qs.total_worker_time,
        qs.statement_start_offset,
        qs.statement_end_offset
    FROM sys.dm_exec_query_stats qs
    ORDER BY qs.total_worker_time desc) qs
CROSS APPLY sys.dm_exec_sql_text(plan_handle) q
CROSS APPLY sys.dm_exec_query_plan(plan_handle) qp
ORDER BY qs.total_worker_time DESC;

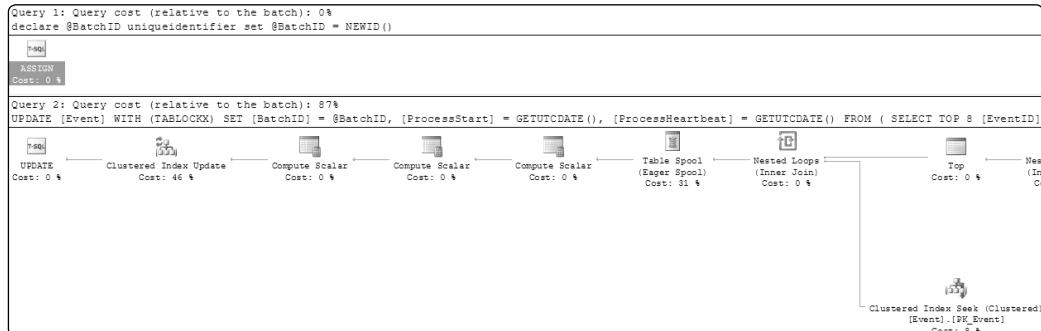
```

---

text	ifade_metni	last_execution_time	execution_count	toplam_cpu_zaman_sn	avg_cpu_zaman_ms
1 declare @BatchID uniqueidentifi...	UPDATE [Event] WITH (TABLOCKX)	2013-02-15 15:41:18.780	1777	0	0
2 declare @BatchID uniqueidentifi...	UPDATE [Notifications] WITH (TABLOCKX)	2013-02-15 15:41:18.780	1777	0	0
3 declare @BatchID uniqueidentifi...	select top 8 - Notification data	2013-02-15 15:41:18.780	1777	0	0
4 SELECT CP.uscounts, ST.text, GP.query_plan, CP.c...	SELECT CP.uscounts, ST.text, GP.query_plan, CP.c...	2013-02-15 15:34:43.443	1	0	261
5 (@_msparam_0 nvarchar(4000) @_msparam_1 nvarchar(400...	SELECT SCHEMA_NAME(ip.schema_id) AS [Schema], sp...	2013-02-15 15:30:37.083	1	0	216
6 @_msparam_0 nvarchar(4000)SELECT CAST(COLLATION...	SELECT CAST(COLLATIONPROPERTY(name, LCID) AS I...	2013-02-15 15:41:14.020	12	0	17
7 @_msparam_0 nvarchar(4000)SELECT SCHEMA_NAME(u...	SELECT SCHEMA_NAME(udf.schema_id) AS [Schema], u...	2013-02-15 15:30:35.507	1	0	204

Sorgu sonucunda hangi sorguların kaç kez çalıştırıldığı, çalışma süresi ve sorgunun SQL kodları gibi bir çok farklı parametre hakkında bilgi alabiliyoruz.

Bu sonuçlarda herhangi bir kaydın sorgu planına ulaşmak için `query_plan` sütunundaki XML veriye Mouse ile tıklanması yeterli olacaktır. Aşağıdaki gibi bir ekran görüntülenir.



SQL Server'da bir başka en çok ihtiyaç duyulan istatistik ise, en çok I/O işlemi yapan sorguların bulunmasıdır. Bu sorgular üzerinde performans iyileştirmeleri yapmak için incelemeler gerçekleştirilir.

En fazla I/O gerçekleştiren ilk 20 sorguyu listeleyelim.

---

```

SELECT
    q.[text],
    SUBSTRING(q.text, (qs.statement_start_offset/2)+1,
        ((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(q.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2) + 1) AS ifade_metni,
    qs.last_execution_time,
    qs.execution_count,
    qs.total_logical_reads AS toplam_mantiksal_okuma,
    qs.total_logical_reads/execution_count AS avg_mantiksal_okuma,
    qs.total_worker_time/1000000 AS total_cpu_time_sn,
    qs.total_worker_time/qs.execution_count/1000 AS avg_cpu_zaman_ms,
    qp.query_plan,
    DB_NAME(q.dbid) AS veritabani_ismi,
    q.objectid,

```

```

        q.number,
        q.encrypted
FROM
    (SELECT TOP 20
        qs.last_execution_time,
        qs.execution_count,
        qs.plan_handle,
        qs.total_worker_time,
        qs.total_logical_reads,
        qs.statement_start_offset,
        qs.statement_end_offset
    FROM sys.dm_exec_query_stats qs
    ORDER BY qs.total_worker_time desc) qs
CROSS APPLY sys.dm_exec_sql_text(plan_handle) q
CROSS APPLY sys.dm_exec_query_plan(plan_handle) qp
ORDER BY qs.total_logical_reads DESC;

```

text	#ade_metri	last_execution_time	execution_count	toplam_mantiksal_okuma	avg_mantiksal_okuma	total_cpu_time_sn
1 (@_mparam_0 nvarchar(4000) @_mparam_1 nvarchar(400...)	SELECT SCHEMA_NAME(sp.schema_id) AS [Schema], sp...	2013-02-15 15:30:45.720	1	80947	80947	0
2 @_mparam_0 nvarchar(4000) @_mparam_1 nvarchar(400...)	SELECT SCHEMA_NAME(sp.schema_id) AS [Schema], sp...	2013-02-15 15:30:37.083	1	80442	80442	0
3 @_mparam_0 nvarchar(4000) @_mparam_1 nvarchar(400...)	SELECT SCHEMA_NAME(sp.schema_id) AS [Schema], sp...	2013-02-15 15:37:45.563	1	80403	80403	0
4 @_mparam_0 nvarchar(4000) SELECT SCHEMA_NAME(u...	SELECT SCHEMA_NAME(jud(schema_id) AS [Schema], ud...	2013-02-15 15:30:35.507	1	36390	36390	0
5 @_mparam_0 nvarchar(4000)SELECT SCHEMA_NAME(v...	SELECT SCHEMA_NAME(v.schema_id) AS [Schema], v.na...	2013-02-15 15:30:34.983	1	17246	17246	0
6 declare @BatchID uniqueidentifier ...	select top 8 - Notification data	2013-02-15 15:43:58.783	1793	14428	8	0
7 @_mparam_0 nvarchar(4000) @_mparam_1 nvarchar(400...	SELECT column_name AS [Name], column_id AS [ID],...	2013-02-15 15:41:14.487	210	6049	28	0

Sorgu sonucunda en çok I/O işlemi yapan ilk 20 sorgu listelendi. Bilgiler arasında sorgunun ne kadar çalıştırıldığı, ne kadar I/O işlemi yaptığı (**total\_logical\_read**) ve işlemin ne kadar sürdüğü gibi bilgiler yer alır.

Bu tür sorgular, sorguların performansını artırmak için çok önemlidir. Yüksek kaynak tüketen sorgularda genel olarak, performansı olumsuz etkileyebilecek sorgu kullanımı mevcuttur. Bu yanlış geliştirmeden kaynaklı sorguları düzenlemek performansı artıracaktır.

## İŞLEMLERDEKİ BEKLEME SORUNU HAKKINDA BİLGİ ALMAK

İşlemler birçok durumda beklemeler yapar. Sebebini tam olarak anlayabilmek için, bazı sistem özelliklerinden bilgi almak gereklidir. Network I/O işlemlerinde mi bekleme var yoksa CPU gibi donanım yetersizliği nedeniyle mi? Bunu anlayabilmek için **sys.dm\_os\_wait\_stats** isimli DMV kullanılır.

---

```
SELECT * FROM sys.dm_os_wait_stats;
```

---

wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1 MISCELLANEOUS	0	0	0	0
2 LCK_M_SCH_S	0	0	0	0
3 LCK_M_SCH_M	11	9	2	0
4 LCK_M_S	14	223482	37195	2
5 LCK_M_U	0	0	0	0
6 LCK_M_X	0	0	0	0
7 LCK_M_IS	0	0	0	0

Sorgu sonucunda dönen sütunları inceleyelim.

- **wait\_type**: Bekleme tipi.
- **waiting\_tasks\_count**: Bu bekleme ile kaç kez karşılaşıldı.
- **wait\_time\_ms**: Toplam bekleme süresi (milisaniye).
- **max\_wait\_time\_ms**: Bekleme tipi için gerçekleşen maksimum bekleme süresi.
- **signal\_wait\_time\_ms**: Bekleme tipinin signaled edildiği zaman ile çalışmaya başladığı zaman arasında geçen sürenin toplamı.

`sys.dm_os_wait_stats` isimli DMV, kümülatif olarak veri toplayan bir DMV'dir. Bellekte çalışan bu tür bir view'in topladığı verileri temizleme ve yeniden istatistiksel bilgiler toplamaya başlaması sağlanabilir. Bunun için iki yöntem vardır. Bunlar, SQL Server servisinin restart edilmesiyle bellek üzerindeki tüm işlemlerin sıfırlanması ya da aşağıdaki gibi topladığı veriyi temizlemekti.

---

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
```

---

```
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

## DİSK CEVAP SÜRESİ HAKKINDA BİLGİ ALMAK

Performans için önemli noktalardan biri de **Disk Response Time** (*disk cevap süresi*)'dır. SQL Server isteğine, disk'in ne kadar gecikmeli olarak cevap verdiği gösteren bir ifadedir. SQL Server açısından önerilen değer 10 milisaniyenin altında olmasıdır.

Bu bilgiyi bize sunan DMV'nin adı `sys.dm_io_virtual_file_stats`'dır.

Bu DMV iki parametre alır. Bunlar;

- **DB ID**: Sadece ID değeri verilen veritabanının dosyalarını sorgular.
- **File ID**: Bir veritabanının sadece belirli dosyalarının response time'larına bakmak için kullanılır. Örneğin; bir veritabanının sadece log file'ının response time'larına bakmak için log file'in **File ID** değeri parametre olarak verilir.

---

```
SELECT * FROM sys.dm_io_virtual_file_stats(null, null);
```

---

Parametreleri **NULL** olarak belirterek sorguladığınızda aşağıdaki gibi bir sonuç listelenenecektir.

database_id	file_id	sample_ms	num_of_reads	num_of_bytes_read	io_stall_read_ms	num_of_writes	num_of_bytes_written	io_stall_write_ms	io_stall	size_on_disk_bytes	file_handle	
1	1	18217952	55	3457024	642	0	0	0	642	5111808	0x0000000000000076C	
2	1	18217952	10	372736	26	12	49152	46	72	1835008	0x00000000000000770	
3	2	18217952	31	1851392	12	2	16384	1	13	8388608	0x00000000000000C10	
4	2	18217952	5	503808	34	4	135168	3	37	524288	0x00000000000000C0C	
5	3	18217952	80	5070848	18	2	16384	1	19	4259840	0x00000000000000C04	
6	3	2	18217952	9	516096	18	8	40960	3	21	1310720	0x00000000000000C08
7	4	18217952	66	4087808	146	1	8192	2	148	17498112	0x00000000000000BA8	

**database\_id**: Veritabanı ID'si.

- **file\_id**: File'ın ID'si.
- **sample\_ms**: SQL Server'ın en son başladığı andan itibaren kaç milisaniye geçtiğini gösterir.
- **num\_of\_reads**: Dosyadan kaç kez okuma işlemi yapıldığını gösterir.
- **num\_of\_bytes\_read**: Yapılan bu okumalarda toplam kaç bayt veri okunduğunu gösterir.
- **io\_stall\_read\_ms**: Yapılan okuma işlemlerinde kullanıcının ne kadar beklediğini gösterir.
- **io\_stall\_write\_ms**: Yapılan yazma işlemlerinde kullanıcının ne kadar beklediğini gösterir.
- **io\_stall**: Okuma ve yazma işlemi için toplam bekleme süresini gösterir.
- **num\_of\_writes**: Dosyaya kaç kez yazma işlemi yapıldığını gösterir.
- **num\_of\_bytes\_write**: Yazma işlemlerinde toplam kaç bayt veri yazıldığını gösterir.
- **size\_on\_disk\_bytes**: Dosya'nın içerisinde gerçekten kullanılmakta olan kısmının bayt cinsinden ifade eder.

**AdventureWorks** veritabanı dosyalarının okuma, yazma gibi istatistiklerini gösterecek bir sorgu hazırlayalım.

---

```
SELECT db_name(mf.database_id) AS veritabani_ismi,
       mf.name AS mantiksal_dosya_ismi,
       io_stall_read_ms/num_of_reads AS cevap_okuma_suresi,
       io_stall_write_ms/num_of_writes AS cevap_yazma_suresi,
       io_stall/(num_of_reads+num_of_writes) AS cevap_suresi,
       num_of_reads, num_of_bytes_read, io_stall_read_ms,
       num_of_writes, num_of_bytes_written, io_stall_write_ms,
       io_stall, size_on_disk_bytes
  FROM sys.dm_io_virtual_file_stats(DB_ID('AdventureWorks'), NULL) AS divFS
 JOIN sys.master_files AS MF
   ON MF.database_id = divFS.database_id
  AND MF.file_id = divFS.file_id;
```

---

	veritabani_ismi	mantiksal_dosya_ismi	cevap_okuma_suresi	cevap_yazma_suresi	num_of_reads	num_of_bytes_read	io_stall_read_ms	num_of_writes	num_of_bytes_written	io_stall_write_ms	io_stall
1	AdventureWorks2012	AdventureWorks2012_Data	25	0	25	88	5578752	2255	1	8192	0
2	AdventureWorks2012	AdventureWorks2012_Log	296	0	110	10	471040	2960	17	73728	13

Veritabanının veri ve loglarını tutan MDF ve LDF uzantılı dosyaları hakkında istatistikleri elde ettik.

## BEKLEYEN I/O İSTEKLERİ HAKKINDA BİLGİ ALMAK

Bekleme durumundaki I/O istekleri hakkında bilgi almak için **sys.dm\_io\_pending\_io\_requests** isimli DMV kullanılır.

---

```
SELECT * FROM sys.dm_io_pending_io_requests;
```

---

	io_completion_request_address	io_type	io_pending_ms_ticks	io_pending	io_completion_routine_address	io_user_data_address	scheduler_address	io_handle	io_offset
1	0x000000037F5B83E8	network	1819226	1	0x000007FEF2F11C50	0x000000037F5B83E8	0x000000037F140D40	0x0000000000000AB0	0

Bu DMV ile bekleme (*pending*) durumundaki I/O istekleri getirilir. Dolayısıyla, sorgu sonucu boş olarak gelirse, beklemeye olan bir I/O isteği olmadığı anlamına gelecektir.

Sorgu sonucunda gelen sütunları inceleyelim.

- **io\_type:** I/O isteğinın şuan hangi aşamada bekleme durumunda olduğunu gösterir.

- **io\_pending\_ms\_ticks:** Toplam ne kadar süredir (*milisaniye*) bekleme durumunda olduğunu belirtir.
- **io\_pending:** İsteğin gerçekten bekleme durumunda mı, yoksa işlem bitirilmiş olmasına rağmen SQL Server'a ulaştırılmadı mı durumunu gösterir. Değer 1 ise istek beklemede, 0 ise istek tamamlanmış ama SQL Server bu bilgiyi almamıştır.
- **io\_handle:** I/O handle'ı.

Beklemede olan I/O isteklerini getirelim.

---

```
SELECT
    FS.database_id AS veritabani_id,
    db_name(FS.database_id) AS veritabani_ismi,
    MF.name AS logical_file_name,
    IP.io_type,
    IP.io_pending_ms_ticks,
    IP.io_pending
FROM sys.dm_io_pending_io_requests IP
LEFT JOIN sys.dm_io_virtual_file_stats(null, null) FS
    ON FS.file_handle = IP.io_handle
LEFT JOIN sys.master_files MF
    ON MF.database_id = FS.database_id
    AND MF.file_id = FS.file_id
```

---

	veritabani_id	veritabani_ismi	logical_file_name	io_type	io_pending_ms_ticks	io_pending
1	NULL	NULL	NULL	network	18217617	1

I/O istekleri ve bunların hızlı bir şekilde karşılanması performans açısından çok önemlidir. Bir SQL Server DBA'ın başlıca görevlerinden biri bu performans parametrelerini takip etmektir.

## DBCC SQLPERF İLE DMV İSTATİSTİKLERİNİ TEMİZLEMEK

DMV'ler, SQL Server servisi başlatıldıktan sonraki istatistikleri toplayan ve servis yeniden başlatılınca, bellekteki bu verileri temizleyen kümülatif veri tutan yapılardır.

DMV'lerin tuttuğu verileri servisi yeniden başlatmadan da temizlemek mümkündür. Bunun için DBCC SQLPERF kullanılabilir.

`sys.dm_os_wait_stats` isimli DMV verisini, `wait_time_ms` değerine göre listeleyelim.

---

```
SELECT * FROM sys.dm_os_wait_stats
WHERE wait_time_ms > 0;
```

---

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PAGEIOLATCH_SH	1	80	80	0
2	LAZYWRITER_SLEEP	188	188458	1016	0
3	ASYNC_NETWORK_IO	1466	433	14	97
4	SLEEP_TASK	513	95812	1027	1
5	SOS_SCHEDULER_YIELD	46756	124	1	93
6	LOGMGR_QUEUE	1455	189154	136	4
7	REQUEST_FOR_DEADLOCK_SEARCH	38	189992	4999	189992

---

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
```

---

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Tekrar aynı listeleme sorgusu çalıştırıldığında, DMV'nin temizlenmiş olduğu görülecektir.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	LAZYWRITER_SLEEP	27	27065	1002	0
2	ASYNC_NETWORK_IO	232	46	3	7
3	SLEEP_TASK	68	13325	1025	0
4	SOS_SCHEDULER_YIELD	6573	13	0	9
5	LOGMGR_QUEUE	205	26651	133	0
6	REQUEST_FOR_DEADLOCK_SEARCH	6	29998	4999	29998
7	SQLTRACE_INCREMENTAL_FLUSH_SLEEP	6	24014	4002	0

Siz de, örnekteki yöntemi kullanarak `sys.dm_os_latch_stats` isimli ya da bir başka DMV'nin tuttuğu veriyi listeleyerek DBCC SQLPERF ile temizleyebilirsiniz.

## STORED PROCEDURE İSTATİSTİKLERİ HAKKINDA BİLGİ ALMAK

Stored Procedure'lerin CPU ve I/O gibi kaynak tüketimleri hakkında bilgi alarak performans analizleri gerçekleştirmeyi sağlayan DMV'nin ismi `sys.dm_exec_procedure_stats`'dır.

---

```
SELECT * FROM sys.dm_exec_procedure_stats;
```

---

plan_handle	cached_time	last_execution_time	execution_count	total_worker_time	last_worker_time	min_worker_time	max_worker_time
0x0500050010148551F0D7E17F03000000010000000000000000...	2013-02-15 10:43:51.687	2013-02-15 10:44:02.983	2	191	48	48	142
0x05000500A659313F10194F720300000010000000000000000...	2013-02-15 10:54:03.330	2013-02-15 15:44:03.473	30	5442	88	63	375
0x050005007E441831D0DE17F0300000010000000000000000...	2013-02-15 10:43:51.503	2013-02-15 10:44:02.970	3	230	47	47	114
0x05000500F4A4555530BC4574D300000010000000000000000...	2013-02-15 10:54:03.177	2013-02-15 15:44:03.467	30	6073	107	52	328
0x0500050062D5E834D0454574D300000010000000000000000...	2013-02-15 10:44:03.177	2013-02-15 15:44:03.487	21	3837	56	56	272
0x050005006461134610DFE17F0300000010000000000000000...	2013-02-15 10:44:03.147	2013-02-15 15:44:03.487	21	5207	109	96	419
0x050005000133EA44301F4F720300000010000000000000000...	2013-02-15 10:54:03.347	2013-02-15 15:44:03.473	30	59616	1149	761	3702

En çok CPU kaynağı harcayan ilk 20 Stored Procedure'ü listeleyelim.

SQL Server'da çalıştırılan Stored Procedure'leri takip etmek, performans analizleri ve prosedürlerin performanslarını artırmak için gereklidir. Performans ve güvenlik nedeniyle kullanılan prosedürlerin performansını takip etmek için **sys.dm\_exec\_procedure\_stats** isimli DMV kullanılır.

---

```
SELECT TOP 20
    DB_NAME(database_id) AS DB_ismi,
    OBJECT_NAME(object_id) AS SP_ismi,
    st.[text] AS SP_kod,
    qp.query_plan,
    cached_time AS ilk_calisma_zamani,
    last_execution_time,
    execution_count,
    ps.total_logical_reads AS toplam_mantiksal_okuma,
    ps.total_logical_reads / execution_count AS avg_mantiksal_okuma,
    ps.total_worker_time / 1000 AS toplam_cpu_zamani_ms,
    ps.total_worker_time / ps.execution_count/1000 AS avg_cpu_zamani_ms
FROM sys.dm_exec_procedure_stats ps
CROSS APPLY sys.dm_exec_sql_text(ps.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(ps.plan_handle) qp
WHERE DB_NAME(database_id)='AdventureWorks'
ORDER BY ps.total_worker_time DESC;
```

---

AdventureWorks veritabanını kullanarak bu sorguyu çalıştırıldığım için sorgu sonucu boş geldi. Bunun nedeni; bilgisayarı başlattığımdan beri AdventureWorks veritabanında hiç bir prosedür çalıştırılmış olmamıdır. DMV'ler servis başlatıldıktan sonra bu bilgileri tutuyordu. Farklı birkaç AdventureWorks veritabanı prosedürü çalıştırılarak prosedür istatistiklerine ulaşılabilir.

İki prosedür çalıştırarak istatistiklere ulaşalım.

DB_ismi	SP_ismi	SP_kod	query_plan	ilk_calisma_zamani	last_execution_time	execution_count
1 AdventureWorks2012	pr_UrunAra	CREATE PROCEDURE pr_UrunAra @ProductName VARCHAR...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	2013-02-15 15:53:15.933	2013-02-15 15:54:08.023	9
2 AdventureWorks2012	pr_ProcedureCall	CREATE PROCEDURE pr_ProcedureCall( @sp_id VARCHAR(2000) ...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	2013-02-15 15:54:04.560	2013-02-15 15:54:05.490	3

toplam_mantiksal_okuma	avg_mantiksal_okuma	toplam_cpu_zamani_ms	avg_cpu_zamani_ms
624	69	512	56
48	16	172	57

## KULLANILMAYAN STORED PROCEDURE'LERİN TESPİT EDİLMESİ

Veritabanın için geliştirilen nesnelerin tamamı sürekli kullanılmayabilir. Bir geliştirme sırasında test için oluşturulan stored procedure ya da kendisini kullanan uygulamanın arayüzü iptal edilen ve artık gerek duyulmayan bir stored procedure olabilir. Bunlar zaman içerisinde gerçekleşebilecek olası düzenleme ihtiyaçlarını doğurur.

Kullanılmayan bir Stored Procedure'ü bulmak için de `sys.dm_exec_procedure_stats` isimli DMV kullanılabilir. Ancak, bu konuda dikkat edilmesi gereken husus var. DMV'ler disk değil, bellek üzerinde kümülatif olarak veri tutuyorlardı. Yani, bir DMV'nin tuttuğu veri veritabanı oluşturulmasından şu ana kadar değil, servis başlatıldığından şimdiki zamana kadar olan istatistiklerdir. Bu durumda bir prosedürün gerçekten kullanılmış kullanılmadığını anlayabilmek için SQL Server servisinin uzun zamandır çalışıyor olduğuna emin olunmalıdır. Ayrıca, her prosedür günlük ya da haftalık kullanılmaz. Özel görevleri olan prosedürler aylık ya da 6 aylık periyotlar ile kullanılıyor da olabilir.

`AdventureWorks` veritabanında servis başlatıldıktan sonra kullanılmayan prosedürleri listeleyelim.

---

```

SELECT SCHEMA_NAME(O.schema_id) AS sema_ismi,
       P.name AS nesne_ismi,
       P.create_date,
       P.modify_date
  FROM sys.procedures P
 LEFT JOIN sys.objects O ON P.object_id = O.object_id
 WHERE P.type = 'P'
 AND NOT EXISTS(SELECT PS.object_id
                  FROM sys.dm_exec_procedure_stats PS
                 WHERE PS.object_id = P.object_id
                   AND PS.database_id=DB_ID('AdventureWorks'))
 ORDER BY SCHEMA_NAME(O.schema_id), P.name;

```

---

	sema_ismi	nesne_ismi	create_date	modify_date
1	dbo	AdayEkle	2013-02-08 02:51:23.963	2013-02-08 02:51:23.963
2	dbo	Gsp_Create_GenericCursor	2013-02-02 23:47:27.533	2013-02-02 23:47:27.533
3	dbo	KitapEkle	2013-02-08 04:28:24.383	2013-02-08 04:28:24.383
4	dbo	pr_CreateDB	2013-02-03 02:00:34.637	2013-02-03 02:00:34.637
5	dbo	pr_HataBilgisiGetir	2013-02-05 13:07:51.123	2013-02-05 13:07:51.123
6	dbo	pr_HataGoster	2013-02-05 13:35:04.720	2013-02-05 13:35:04.720
7	dbo	pr_InsertLocation	2013-02-04 22:06:19.727	2013-02-04 22:34:29.737

Bu sorgu sonucunda gelen liste üzerinde, veritabanı programcısı ya da SQL Server DBA'ı inceleme yaparak, hangi prosedürlerin kullanım ömrünü doldurduğunu bilebilir. Sonuç olarak, geliştirilen prosedürler ihtiyaçlar doğrultusunda DBA ya da veritabanı programcısı tarafından oluşturulmaktadır.

## TRIGGER İSTATİSTİKLERİ

Trigger bölümünde ve trigger ile ilgili performans konularını incelediğimiz bu bölümde, trigger'in tetiklendiği transaction'ın bir parçası olduğundan bahsettik. Tetikleyici ve tetiklenen trigger arasında bir performans ortaklısı vardır. Bir Stored Procedure içerisindeki transaction'ın tamamlanması için, tetiklediği trigger'in ta **ROLLBACK** olmadan tamamlanması gereklidir. Bu durumda, birinin performansı diğerini etkileyecektir. Bu performans ortaklığının önemini kavradıktan sonra, trigger ile ilgili istatistikleri inceleyebiliriz.

Trigger istatistiklerini tutan DMV'nin ismi **sys.dm\_exec\_trigger\_stats**'dır.

---

```
select * from sys.dm_exec_trigger_stats;
```

---

En çok CPU tüketen ilk 10 trigger'ı listeleyelim.

---

```
SELECT TOP 10
    DB_NAME(database_id) AS DB_ismi,
    SCHEMA_NAME(O.schema_id) AS tablo_sema_ismi,
    O.name AS tablo_nesne_ismi,
    T.name AS trigger_nesne_ismi,
    ST.[text] AS trigger_kod,
    QP.query_plan,
    cached_time AS ilk_calisma_zamani,
    last_execution_time,
    execution_count,
    PS.total_logical_reads AS toplam_mantiksal_okuma,
```

```

PS.total_logical_reads / execution_count AS avg_mantiksal_okuma,
PS.total_worker_time / 1000 AS toplam_cpu_zaman_ms,
PS.total_worker_time / PS.execution_count / 1000 AS avg_cpu_zaman_ms
FROM sys.dm_exec_trigger_stats PS
LEFT JOIN sys.triggers T ON T.object_id = PS.object_id
LEFT JOIN sys.objects O ON O.object_id = T.parent_id
CROSS APPLY sys.dm_exec_sql_text(PS.plan_handle) ST
CROSS APPLY sys.dm_exec_query_plan(PS.plan_handle) QP
WHERE DB_NAME(database_id) = 'AdventureWorks'
ORDER BY PS.total_worker_time DESC;

```

---

Servisin son başladığı andan itibaren veri tutulmaya başlandığı için bu sorgu sonucunda hiç bir kayıt dönmemesi bizi şaşırtmamalı.

Örnek bir kaç trigger çalıştırarak yukarıdaki soruyu tekrar inceleyelim.

**Production.WorkOrder** tablosu üzerinde bir güncelleştirme yapalım.

---

```

UPDATE Production.WorkOrder
SET StartDate = GETDATE(), EndDate = GETDATE(), DueDate = GETDATE()
WHERE WorkOrderID = 1

```

---

**Sales.SalesOrderHeader** tablosu üzerinde bir güncelleştirme yapalım.

---

```

UPDATE Sales.SalesOrderHeader
SET OrderDate = GETDATE(), DueDate = GETDATE(), ShipDate = GETDATE()
WHERE SalesOrderID = 75123;

```

---

Bu sorgular sonucunda **UPDATE** komutunu takip eden trigger'lar tetiklenecek ve bizim takip ettiğimiz istatistikler tarafından görünür hale gelecektir.

Yukarıdaki trigger istatistik sorgusu tekrar çalıştırıldığında istenen istatistikler elde edilecektir.

DB_İsmi	tablo_sema_İsmi	tablo_nane_İsmi	trigger_nane_İsmi	trigger_kod	query_plan	İk_çalışma_zamani
1 AdventureWorks2012	Sales	SalesOrderHeader	uSalesOrderHeader	CREATE TRIGGER [Sales].[uSalesOrderHeader] ON ...	<a href="#">ShowPlanXML xmlns='http://schemas.microsoft.com...'</a>	2013-02-15 15:58:09.673
2 AdventureWorks2012	Production	WorkOrder	uWorkOrder	CREATE TRIGGER [Production].[uWorkOrder] ON [P...]	<a href="#">ShowPlanXML xmlns='http://schemas.microsoft.com...'</a>	2013-02-15 15:58:02.500

last_execution_time	execution_count	toplam_mantiksal_okuma	avg_mantiksal_okuma	toplam_cpu_zaman_ms	avg_cpu_zaman_ms
2013-02-15 15:58:10.400	2	10	5	0	0
2013-02-15 15:58:04.750	5	0	0	0	0

# KULLANILMAYAN TRIGGER'LARI

## TESPİT ETMEK

Kullanılmayan Stored Procedure'leri incelerken bahsettiğimiz durumların tamamı trigger'lar için de geçerlidir. Veritabanı geliştirildikten sonra, sürekli değişen ihtiyaçlar doğrultusunda, bazen daha önceden geliştirilmiş trigger'lar artık kullanılmamaya başlanabilir. Bu tür kullanılmayan trigger'lar takip edilmeli ve gerekiyorsa DROP edilmelidir.

---

```

SELECT SCHEMA_NAME(o.schema_id) AS tablo_sema_ismi,
       o.name AS tablo_nesne_ismi,
       t.name AS trigger_nesne_ismi,
       t.create_date,
       t.modify_date
  FROM sys.triggers t
 LEFT JOIN sys.objects o ON t.parent_id = o.object_id
 WHERE t.is_disabled = 0 AND t.parent_id > 0
AND NOT EXISTS(SELECT ps.object_id
   FROM sys.dm_exec_procedure_stats ps
  WHERE ps.object_id = t.object_id
    AND ps.database_id = DB_ID('AdventureWorks'))
 ORDER BY SCHEMA_NAME(o.schema_id),o.name,t.name;

```

---

	tablo_sema_ismi	tablo_nesne_ismi	trigger_nesne_ismi	create_date	modify_date
1	dbo	Calisanlar	trgAfterInsert	2013-02-11 12:18:36.620	2013-02-11 12:18:36.620
2	dbo	Kullanicilar	trg_KullanicilSil	2013-02-11 22:34:56.673	2013-02-11 22:34:56.673
3	dbo	Makaleler	trg_MakaleSil	2013-02-11 21:14:32.040	2013-02-11 21:14:32.040
4	dbo	Personeller	trg_PersonelHatirlatici	2013-02-13 01:58:44.880	2013-02-13 01:58:44.880
5	dbo	vw_MusteriSiparisleri	trg_MusteriSiparisEkle	2013-02-13 10:36:56.003	2013-02-13 10:36:56.003
6	dbo	vw_MusteriSiparisleri	trg_MusteriSiparisSil	2013-02-13 10:42:51.687	2013-02-13 10:42:51.687
7	HumanResources	Employee	dEmployee	2012-03-14 13:14:55.383	2012-03-14 13:14:55.383

Sorgu sonucunda dönen kayıtlar, **AdventureWorks** veritabanında kullanılmayan trigger'lardır. Tekrar hatırlatmak gerekiyor ki, bu sorgu ile elde edilen kayıtlar SQL Server servisinin başladığı andan itibaren kullanılmayan trigger'lardır. SQL Server 1 gün önce mi başlatıldı, yoksa 1 yıl mı, bunu ancak SQL Server DBA bilebilir.

## AÇIK OLAN CURSOR'LERİ SORGULAMA

Bu kitabın cursor'ler ile ilgili bölümünde performans ile ilgili önemli bir konuya değinmiştik. Cursor'ler kullanılmak için açılmalı ve işi bitince kapatılmalıdır. Ancak kapatılmadığı takdirde sistem kaynaklarını tüketmeye devam edecektir. Sistemdeki açık cursor'leri takip edebilmek, bu cursor'leri kimin, ne zaman açtığını ve içerdeği sorguların neler olduğunu öğrenebilmek için `sys.dm_exec_cursors` DMF'si kullanılır.

---

```
SELECT * FROM sys.dm_exec_cursors(0);
```

---

Bu DMF, parametre olarak `Session ID` alır. Varsayılan kullanım 0 parametresidir. Sıfır parametresi ile kullanıldığında, sorgunun çalıştırıldığı veritabanı üzerindeki açık olan tüm cursor'leri listeler. Parametre olarak bir `Session ID` verilirse, bu `Session ID` için açık olan cursor'leri listeler.

Sorgu sonucunda dönen sütunları inceleyelim.

- `session_id`: Cursor'ın hangi session üzerinde açıldığını belirtir.
- `cursor_id`: Cursor'ın ID bilgisi.
- `name`: Cursor'ın ismi.
- `properties`: Cursor'ın hangi özellikler ile açıldığını belirtir.
- `sql_handle`: Cursor'ın `sql_handle`'ıdır.
- `creation_time`: Cursor'ın oluşturulma zamanını belirtir.
- `fetch_status`: Cursor'ın en son `FETCH` durumunu belirtir.
- `dormant_duration`: En son `OPEN` ya da `FETCH` işleminden sonra geçen zamanı (*milisaniye*) belirtir.

Şuan sisteminizde açık cursor olmadığını varsayarak yeni bir cursor tanımlamalıyız. Şimdi, örnek olarak, belirli bir süre açık kalacak cursor tanımlayalım.

---

```
DECLARE @ProductID INT;
DECLARE @Name VARCHAR(255);
DECLARE ProductCursor CURSOR FOR
    SELECT ProductID, Name FROM Production.Product WHERE ProductID < 5;
OPEN ProductCursor;
```

```

FETCH NEXT FROM ProductCursor INTO @ProductID, @Name;
WHILE @@FETCH_STATUS = 0
BEGIN
WAITFOR DELAY '00:00:2';
PRINT CAST(@ProductID AS VARCHAR) + ' - ' + @Name;
FETCH NEXT FROM ProductCursor INTO @ProductID, @Name;
END;
CLOSE ProductCursor;
DEALLOCATE ProductCursor;

```

---

Cursor açıkken aşağıdaki sorguyu çalıştırarak açık olan cursor'leri bulalım.

```

SELECT ec.cursor_id,
       ec.name AS cursor_name,
       ec.creation_time,
       ec.dormant_duration/1000 AS uyku_suresi_sn,
       ec.fetch_status,
       ec.properties,
       ec.session_id,
       es.login_name,
       es.host_name,
       st.text,
       SUBSTRING(st.text, (ec.statement_start_offset/2)+1,
                  ((CASE ec.statement_end_offset
                      WHEN -1 THEN DATALENGTH(st.text)
                      ELSE ec.statement_end_offset
                  END - ec.statement_start_offset)/2) + 1) AS ifade_metni
FROM sys.dm_exec_cursors(0) ec
CROSS APPLY sys.dm_exec_sql_text(ec.sql_handle) st
JOIN sys.dm_exec_sessions es ON es.session_id = ec.session_id

```

---

	cursor_id	cursor_name	creation_time	uyku_suresi_sn	fetch_status	properties	session_id	login_name	host_name	text
1	180150005	ProductCursor	2013-02-15 16:01:24.480	0	0	TSQL   Dynamic   Optimistic   Global (0)	54	djbil-pc\djibil	DJIBIL-PC	DECLARE @ProductID INT; DECLARE @Name VARCHAR(..

Sorgu sonucuna baktığımızda, cursor id ve isim bilgisi, oluşturulma zamanı, **FETCH** durumu, özellikleri, session'ın id bilgisi, login ve bilgisayar adı ve cursor'ın içерdiği SQL sorgu metni gibi bilgiler listelenmektedir.



# YEDEKLEMEMEK VE YEDEKTEN DÖNMEK

20

Yazılımcıların da zaman zaman gerçekleştirdiği bir görev olan yedekleme işlemi, veritabanı yöneticisi görevlerinde ilk sıralarda yer alır. Veritabanı yöneticisi, yedekleme ve yedekleri yönetme görevini yerine getirir.

Bu bölümde, yedek türleri, veritabanı strateji ve planı oluşturma, SSMS ve T-SQL ile yedekleme işlemini gerçekleştirmek, alınan yedekleri geri yüklemek gibi işlemleri ele alacağız.

## VERİTABANINI YEDEKLEMEMEK

Veritabanı projelerinde en önemli ve korunması gereken öncelikle sistem verinin kendisidir. Veritabanında bulunan verinin korunabilmesi, birçok farklı tehlikeye karşı gerçekleştirilir.

Bu tehlikeleri aşağıdaki gibi sıralayabiliriz.

## YAZILIMSAL SORUNLAR

İnsan ve yazılım faktörünün bulunduğu her yerde olası sorun ve hatalara karşı hazırlıklı olunmalıdır. Veritabanı üzerinde çalışan yazılımların algoritma hataları, veritabanı geliştiricisi ya da yöneticisinin kazara veri silmesi, veritabanına izinsiz erişim ile gerçekleştirilen hacking saldıruları, işletim sistemi hataları, VTYs yönetim yazılıminin hataları gibi bir çok farklı sebepten kaynaklanan hataları kapsar.

## DONANIMSAL SORUNLAR

Veritabanı ve yazılımsal fiziksel donanımlar üzerinde çalışır. Bu durumda, sadece veritabanı ve yazılımdaki hataların giderilmesi tek başına yeterli değildir. Sunucu sistemindeki işlemci, disk ve belki donanımsal RAID sistemindeki olası hataları kapsar.

## FİZİKSEL SORUNLAR

Yazılım ve donanımsal hatalar önemli ve en çok rastlanabilecek olası sorunlar arasındadır. Ancak yazılım ve veritabanının yer aldığı donanımların, gerçek hayatı fiziksel konumları da çeşitli sorunlarla karşılaşabilir. Veri merkezi ya da sunucu sistemin bulunduğu farklı bir mekanın doğal afet, silahlı saldırı (sanal ortamda hacking saldırısının gerçek hayatı karşılığıdır) gibi sorunları kapsar.

Kitapta verdığım bir örneği hatırlayalım. Büyük bir GSM operatörünün veri merkezinde meydana gelen sel baskını sonucu, büyük çapta veri kaybı yaşanmış ve GSM operatörü, veri kaybının kapsamındaki müşterilerinin operatör kullanım ücreti verilerini kaybettiği için müşterilere hediye (!) etmek zorunda kalması doğal afetlere örnek verilebilir.

## SQL SERVER OTOMATİK KURTARMA İŞLEMİ

SQL Server sunucusu, hata ya da elektrik kesilmesi gibi olağan olmayan sebeplerle kapandığında, **Buffer Cache** ve **Log Buffer** gibi bellek üzerinde yer alan veriler kaybedilir. SQL Server servisi tekrar çalışmaya başladığında, loglara kaydedilemeyen verileri kurtarmak için SQL Server otomatik kurtarma işlemi devreye girer. Bu işlemle birlikte, veritabanı kararlı hale getirilmek için gerekli işlemler gerçekleştirilmiş olur.

## VERİTABANI LOGLAMA SEÇENEKLERİ

Veritabanı loglarının önem seviyesi ile alakalı 3 farklı loglama seçeneği vardır. Bunlar, transaction log dosyasına yansıtılacak logların seviyelerini ayarlamak için kullanılır. SQL Server da varsayılan seviye **Full Recovery**'dir. Ancak, gerekli durumlarda bu seçenek değiştirilerek performans ve log veri hacmi yönetilebilir.

## SİMPLE RECOVERY MODEL

Veri kaybı önemli olmayan veritabanları için kullanılabilir. Bu model ile, log dosyası belli bir boyutun üzerine çıkmaz ve veritabanında belli bir zamana dönülemez. Loglar sadece işlemin tamamlandığına dair işaretlenene kadar tutulur ve sonrasında bu loglar silinir. Kritik öneme sahip veritabanlarında kullanılması gereklidir.

## FULL RECOVERY MODEL

Bu seçenek ile, tüm değişimler loglanır ve istenilen zamana dönülebilir. Log veri boyutu olarak en çok yer kaplayan modeldir. Loglar düzenli olarak yedeklenmezse, yedeklenene kadar bekletilir. Bu durumda, log dosyası büyük hacimlere ulaşabilir. Bu seçeneğin tam verimli ve doğru kullanılabilmesi için loglar hatalı olarak yedeklenmelidir. Arada herhangi bir verinin kaybolursa, istenilen zamana dönülemeyecektir. Sağlıklı bir veritabanı modeli için veritabanı yedeklemenin yanı sıra, log dosyaları da düzenli olarak yedeklenmelidir. Bunun için zamanlanmış plan kullanılabilir.

## BULK-LOGGED RECOVERY MODEL

Adından da anlaşılacağı gibi, BULK işlemlerinde tercih edilebilecek bir modeldir. BCP ile yapılan yüklemeler, `SELECT INTO` gibi toplu işlemler loglanmaz. Full Recovery moduna sahip sistemde, log dosyasının büyümemesi için Full Recovery ile Bulk-Logged arasında geçiş yapılabilir.

Veritabanı loglama seviyesi SSMS ya da T-SQL ile ayarlanabilir.

T-SQL ile loglama seviyesini ayarlamak için aşağıdaki yöntem kullanılabilir.

---

```
USE master
GO
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
```

---

SSMS ile loglama seviyesini ayarlamak için aşağıdaki yöntem kullanılabilir.

- Ayar yapmak için ilgili veritabanının üzerine fare ile sağ tıklayın.
- Properties menüsünden veritabanı özellikleri ekranına geçin.
- Options menüsü ile veritabanı ayarlarına girin.



## YEDEKLEME TÜRLERİ

SQL Server farklı veritabanı yedekleme yöntemleri ile verinin yedeklenmesi gerçekleştirilebilir. Bu yedekleme yöntemleri aşağıdaki gibi üçe ayrılır.

### TAM VERİTABANI YEDEĞİ (FULL DATABASE BACKUP)

Tüm veritabanının yedeğini almak için kullanılır. Yedekleme için çok kullanılan yöntemdir. Bir veritabanını Full Backup olarak yedekledikten sonra, tek bir yedek dosyası ile veritabanı tekrar kullanılabilir hale getirilebilir. Ancak, herhangi bir zamana geri dönmek için tek başına yeterli değildir.

### FARK YEDEĞİ (DIFFERENTIAL BACKUP)

Tam yedeklemeden sonra yapılan değişikliklerin yedeğini almak için kullanılır. Bir veritabanının tam yedeği alındıktan sonra, her defasında tam yedek alınarak büyük veri alanı kaplamak yerine, bir kez ya da uzun aralıklarla tam yedek alınarak, sonraki yedeklerin tam yedek ile sonraki zaman arasındaki fark yedeği olması sağlanabilir.

Büyük hacimli bir veritabanında aylık tam yedek alındığı bir senaryo için durum şöyle gerçekleştirilebilir. Ayın son günü alınan tam yedekten sonra, her gün için bir fark yedeği alınır. Daha sonra bu fark yedeğinin bulunduğu günün verisine dönülmek istendiğinde tam yedek ve ilgili günün fark yedeği yeterli olacaktır.

### LOG YEDEĞİ (LOG BACKUP)

Veritabanında belli bir zamana donebilmek için log yedekleri büyük öneme sahiptir. Full Backup işleminden sonraki tüm fark yedeklerinin bulunması gereklidir. Fark yedeklerinin birinin bile bulunmaması durumunda, bu tarihten sonraki log yedeklerinin işlevini yitirmesine neden olur. Özetlemek ve kavramı anlamak için veritabanı yedeği ile log yedeklerinin bir bütünü oluşturduğunun bilinmesinde fayda var. İstenen tarihe donebilmek için loglar büyük öneme sahiptir.

## YEDEKLEME VE KURTARMA PLANI OLUŞTURMAK

Veri güvenliği için veritabanı ve logların yedeklenmesi gereklidir. Yedekleme işlemiyle bir strateji ve planlama olmadan yapılmamalıdır. Strateji ve planlama olmadan yapılan yedekleme ve kurtarma planı, ani veri kayıplarında yapılan plansız yedeklemenin anlaşılması ve veri kaybını geri getiremeye gibi büyük sorunlara yol açar.

Veritabanının başlıca görevlerinden biri olan yedekleme ve kurtarma planı oluşturmak, veritabanı yöneticisini iş hayatında seçkin hale getirebilecek kadar önemlidir. Bir geliştirici ya da veritabanı yöneticisi, diğer birçok alanda ne kadar uzman olursa olsun, yedekleme ve kurtarma için gerekli ileri seviye bilgi ve planlama yeteneğine sahip değilse, iş hayatında büyük sorunlarla karşılaşabilecektir.

### BAŞLANGIÇ YEDEKLEME VE KURTARMA PLANLAMASI

Veritabanı planlama stratejisi geliştirmek ve planlamayı teknik anlamda hazırlamak belirli bir süreç içerisinde gerçekleşir. Planlamanın yapılabilmesi için veri, veritabanı ve donanım alt yapısını iyi tanımak gereklidir. Aynı zamanda, veritabanı hakkında aşağıdaki soruların cevabını bilmek, doğru planlama için gereklidir.

### VERİ NE KADAR ÖNEMLİ?

Veritabanının önemini ve ne sıklıkla yedeklenmesi gerektiğini belirleyen şey verinin kendisidir. Verinin önemi arttıkça, daha sık yedekleme ihtiyacı doğar. Geliştirme ortamındaki bir veritabanını, haftada bir ve log yedeği gerektirmeden yedekleyebilirken, anlık veri işlemi yüksek olan bir veritabanının günde bir ya da iki kez yedeklenmesi gerekebilir. Bazen bu da yeterli gelmeyerek aradaki zaman dilimi için fark yedekleri de almak gerekebilir. Veri önemi arttıkça yedek, fark yedeği, log yedeği ve bu yedeklerin önemi de artmaktadır.

### YEDEKLENEN VERİTABANLARININ TÜRÜ NEDİR?

Yedeklenecek veritabanının türü ve hangi amaç için kullanıldığı planlama stratejisi oluşturmak için önemlidir. Örneğin; sık kullanılan bir veritabanı

ile yedek ya da test olarak kullanılan bir veritabanı kopyasının yedekleme stratejisi farklıdır. Hatta bazı veritabanları için yedekleme gereksinimi bile duyulmamayabilir.

Veritabanı türüne göre kritiklik seviyesi değişecektir. Örneğin; bir kullanıcı veritabanı çok sık güncellenip üzerinde işlem yapılır. Bu durumda, veritabanı ve log yedeklemesi için farklı bir planlama yapılmalıdır. En kötü ihtimalle de veri kaybı gerçekleşir. Ancak, master veritabanı çok sık güncellenmemesine rağmen, bozulma ya da bir hata ile karşılaşırsa sunucu başarısız olur. Bazı durumlarda, master veritabanının onarılması ya da yedeğinin kullanılması gerekir.

### **VERİ NE KADAR ÇABUK KURTARILMALI?**

Veri yedeğinin ne kadar sürede aktif hale getirilmesi gerektiği bilinmelidir. Kritik bir ortamda önemli bir veri yedeğinin aktif edilmesi söz konusu ise, verinin disk okuma hızı bile çok önemlidir. Büyük veri kümelerinde uzun sürecek veri kurtarma senaryosu birçok soruna sebep olabilir.

### **YEDEKLEMESİ GERÇEKLEŞTİRECEK DONANIM VAR MI?**

Yedekleme için ek donanıma ihtiyaç vardır. Sağlıklı ve hızlı veri kurtarma işlemi için hızlı ve güçlü donanımlar olmalıdır. Bu bir yedek sunucusu olabileceği gibi, bir disk, teyp ya da farklı bir aygıtta olabilir.

### **YEDEKLEME İÇİN EN UYGUN ZAMAN NEDİR?**

Yedekleme işlemleri genel olarak yayında olan ve üzerinde sürekli işlem yapılan canlı sistemler üzerinde gerçekleştirilir. Kurumsal veritabanlarında, hafta sonu akşam saatleri işlem yoğunluğunun en az olduğu zamanlardır. Yedekleme işlemi, büyük hacimli verilerin bulunduğu sistemlerde uzun süreceği için yoğunluğun en az olduğu zamanlarda gerçekleştirilir. Ancak, her zaman az yoğun olduğumu durumda yedekleme istenmez. Yoğun veri işlemi olan durumlarda da yedekleme işlemi gerekebilir. Yedekleme planının bu duruma göre hazırlanması, yedeklemeden kaynaklanan olası gecikme ve sorunlara çözüm olacaktır.

## YEDEKLEME, SIKIŞTIRILABİLİR Mİ?

SQL Server, veritabanı yedeklerinin sıkıştırılmasını destekler. Sıkıştırılan veritabanı yedeği disk üzerinde daha az yer kaplar. Daha az yer kaplaması nedeniyle daha az disk I/O işlemine sebep olur. Taşınabilirlik ve veri aktarımı açısından da yedeği sıkıştırmak faydalıdır. Daha az yer kaplaması nedeniyle, yedekleme sunucuları ve kullanılan yedek aygıtları, disk ya da teyp gibi donanımsal kaynaklarda az yer kaplar. Bu nedenle, yedekleme donanımı ihtiyacı azalacağı için, donanım maliyeti de azalacaktır. Ancak, sıkıştırma işlemi disk üzerinde faydalı olsa da, işlemci açısından daha fazla iş yükü getireceği gibi, yoğun işlemci gücü harcanmasına neden olur.

## YEDEKLEMELERİ ALAN DIŞINDA SAKLAMAK GEREKİYOR MU?

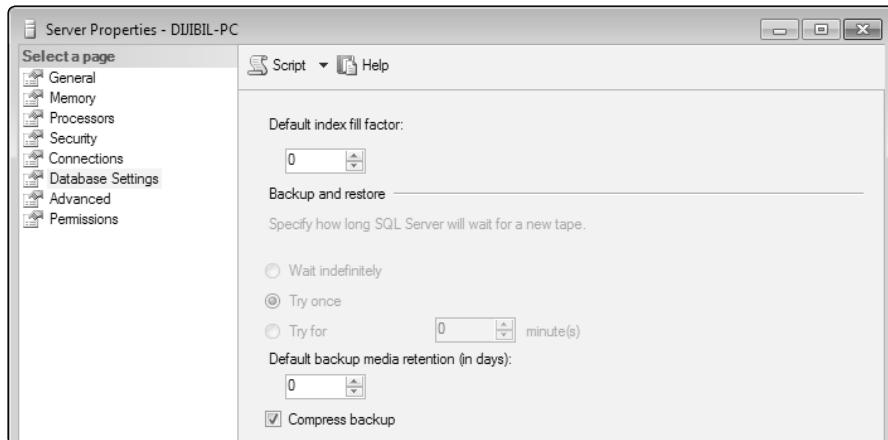
Performans ile ilgili bölümde detaylıca örneklendirdiğimiz bu konu fiziksel sorunlar ile ilgidir. Verinin bulunduğu donanımsal sistemin doğal afet ya da benzeri fiziksel sorunlardan dolayı yayınına devam edememesi durumunda, ikizleme (*aynalama*) ya da yedekten veri kurtarma işlemlerinin gerçekleştirilmesi gereklidir. Kritik verilerin bulunduğu sistemlerin karasal hat olarak farklı bağlantılar, farklı şehir ya da ülkede yedeklerinin bulunması, olası birçok fiziksel sorunun önüne geçer. Fiziksel sorun meydana geldiğinde yedeklenen bu veritabanı ve yazılım yedekleri kullanılarak sistem tekrar ayağa kaldırılabilir.

## YEDEKLEME SIKIŞTIRMASI PLANLAMAK

SQL Server'da yedek sıkıştırma ayarı varsayılan olarak devre dışıdır. Sıkıştırma işlemi iki şekilde gerçekleştirilebilir.

**Management Studio** ile yedek sıkıştırma özelliğini ayarlamak:

- **Management Studio ekranındaki Object Explorer** panelinden sunucu girdisini, farenin sağ düğmesiyle tıklayarak **Server Properties** iletişim kutusunu açın.
- Açılan ekranda **Database Settings** sayfasını açın. Sıkıştırma özelliğini açmak için, **Compress backup** seçeneğini seçili hale getirin. Aynı özelliği kapatmak için, aynı seçenekteki seçeneği kaldırın.



T-SQL ile yedek sıkıştırma özelliğini ayarlamak:

Yedek sıkıştırma ayarları programsal olarak yapılabilir. Bu işlem aşağıdaki gibi yapılabilir.

---

```
EXEC sp_configure 'backup compression default','1';
GO
RECONFIGURE WITH OVERRIDE;
GO
```

---

Yukarıdaki sorgu ile yedek sıkıştırma özelliği aktif edilir. İkinci parametre olarak verilen 1 değeri yerine, 0 değeri verilirse, yedek sıkıştırma özelliği pasif edilir.

Aynı işlemin, hem SSMS hem de T-SQL ile nasıl yapıldığını inceledik. T-SQL ile yapılan ayar değişikliği **Management Studio** ortamındaki seçim kutusunun seçili halinin değişimine sebep olacaktır.

## YEDEKLEMESİ GERÇEKLEŞTİRMEK

Yedekleme işlemini gerçekleştirmek için yedekleme türleri gibi mimari birçok özelliği inceledik. Veritabanı yedekleme işleminin öneminden ve hangi şartlarda ne tür yedeklemeler gerektiği gibi konulara değindik. Artık veritabanı yedekleme işleminin nasıl gerçekleştirileceği incelenebilir.

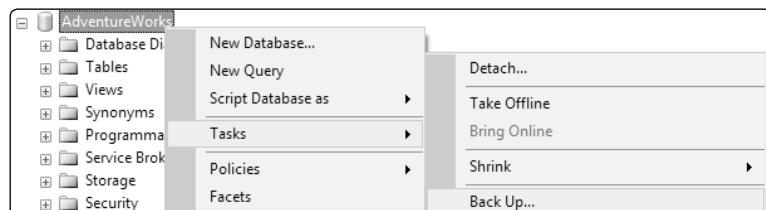
## SQL SERVER MANAGEMENT STUDIO İLE YEDEK OLUŞTURMAK

Yedekleme işlemi, **Management Studio** ile kolay ve hızlı bir şekilde yapılabilir.

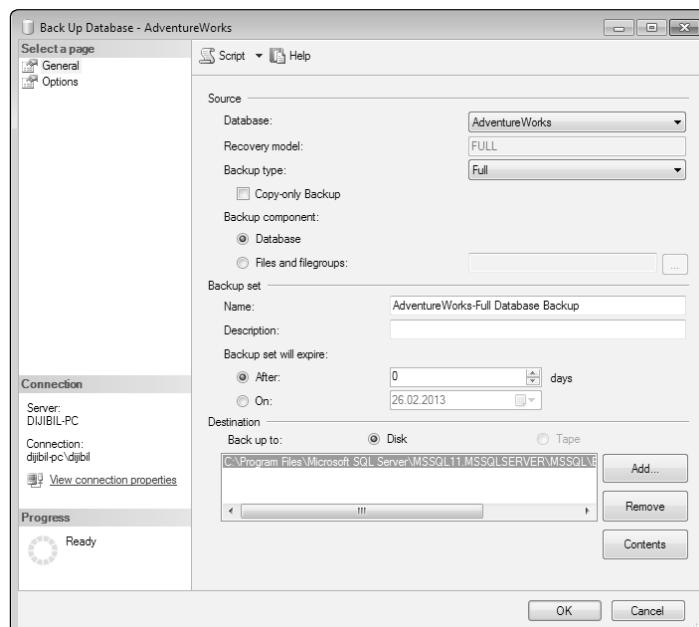
**Management Studio** ile yedekleme işlemi için aşağıdaki adımları takip edin.

**Object Explorer** paneliyle yedeklenmek istenen veritabanının üzerine fare ile sağ tıklayın.

- Açılan pencerede sırasıyla **Tasks > Back Up...** menülerini seçin.
- **Back Up Database** ekranında gerekli ayarlamaları yapıp, **OK** butonuna basılarak yedekleme başlatılabilir.



Yukarıdaki menüler seçildikten sonra, aşağıdaki **Back Up Database** paneli açılacaktır.



**Management Studio** ile oluşturulan yedekler, varsayılan olarak aşağıdaki dosya yolunda saklanır.

C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\Backup

## T-SQL İLE VERİTABANI YEDEĞİ OLUŞTURMAK

T-SQL ile veritabanını yedeklemek için aşağıdaki söz dizimi kullanılabilir.

### Söz Dizimi:

---

```
BACKUP DATABASE veritabani_ismi
TO DISK = 'disk_yolu'[, DISK ='ikinci_disk_yolu',...]
[WITH MEDIANAME='medyaset_ismi'];
```

---

**AdventureWorks** veritabanının tam veritabanı yedeğini (**Full Database Backup**) alalım.

---

```
BACKUP DATABASE AdventureWorks
TO DISK='C:\Backups\AWorks1.bak';
```

---

Sorgunun başarılı çalışabilmesi için dizin olarak belirtilen C:\ içerisinde Backups isimli klasör oluşturulmuş olmalıdır. Aynı zamanda, veritabanının ismi de tam olarak belirtilmelidir.

Bir sonraki alınan yedeğin, var olan dosyanın üstüne yazması için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks
TO DISK='C:\Backups\AWorks1.BAK'
WITH INIT;
```

---

Tam veritabanı yedeği yerine, fark yedeği almak için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks
TO DISK = 'C:\Backups\AWorks1.BAK'
WITH DIFFERENTIAL;
```

---

Fark yedeği almak istendiğinde, fark yedeği sorgusu sık kullanılacaktır.

Veritabanının birden fazla dosyaya yazdırılması için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks
TO DISK='C:\Backups\AWorks1.BAK',
DISK='D:\Backups\AWorks2.BAK',
DISK='E:\Backups\AWorks3.BAK';
```

---

Veritabanının birden fazla parça ayrılması ve bu parçaların birden fazla fiziksel disk üzerinde saklanarak I/O performansı ve hızlı işlem için tercih edilebilecek bir yöntemdir.

Yedeklenen veritabanında, geri yükleme işleminde kullanılacak bir şifre oluşturmak için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks
TO DISK = 'C:\Backups\AWorks1.BAK'
WITH PASSWORD = 'D!J@J#İ$B#İ$L';
```

---

Veritabanı yedekleme işleminin yüzdesel olarak tamamlanma oranının gösterilmesini sağlamak için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks
TO DISK = 'C:\Backups\AWorks4.BAK'
WITH STATS;
```

---

```
10 percent processed.
20 percent processed.
30 percent processed.
40 percent processed.
50 percent processed.
60 percent processed.
70 percent processed.
80 percent processed.
90 percent processed.
Processed 24792 pages for database 'AdventureWorks', file 'AdventureWorks2012_Data' on file 1.
100 percent processed.
Processed 2 pages for database 'AdventureWorks', file 'AdventureWorks2012_Log' on file 1.
BACKUP DATABASE successfully processed 24794 pages in 6.591 seconds (29.389 MB/sec).
```

Yukarıdaki sorgu çalıştırıldığında, **SSMS** sorgu ekranının **Messages** kısmında, varsayılan olarak her tamamlanan **%10** işlem için bildirim yapılır. **WITH STATS** aşağıdaki gibi kullanıldığı taktirde, farklı yüzde oranına göre de mesaj bildirimi yapılabilir.

```
BACKUP DATABASE AdventureWorks  
TO DISK = 'C:\Backups\AWorks5.BAK'  
WITH STATS = 2;
```

---

```
80 percent processed.  
82 percent processed.  
84 percent processed.  
86 percent processed.  
88 percent processed.  
90 percent processed.  
92 percent processed.  
94 percent processed.  
96 percent processed.  
98 percent processed.  
Processed 24792 pages for database 'AdventureWorks', file 'AdventureWorks2012_Data' on file 1.  
100 percent processed.  
Processed 2 pages for database 'AdventureWorks', file 'AdventureWorks2012_Log' on file 1.  
BACKUP DATABASE successfully processed 24794 pages in 7.347 seconds (26.364 MB/sec).
```

Veritabanı yedeği için bir açıklama satırı eklenebilir. Bu işlem için şu ifade kullanılabilir.

---

```
BACKUP DATABASE AdventureWorks  
TO DISK = 'C:\Backups\AWorks6.BAK'  
WITH DESCRIPTION = 'AdventureWorks için Tam Yedek';
```

---

Yedekleme sırasında, birden fazla özellik tanımlayabilmek için şu ifade kullanılabilir. Bu sorguda, hem Mirror işlemi, hem de birden fazla özellik tanımlaması yapalım.

---

```
BACKUP DATABASE AdventureWorks  
TO DISK = 'C:\Backups\AWorks7.BAK'  
MIRROR TO DISK = 'C:\Backups\AdventureWorks_MIRROR.BAK'  
WITH FORMAT, STATS, PASSWORD = 'D!İ@J#İ$B#İ$L';
```

---

Mirror, hazırlanan yedeğin bir kopyasının daha oluşturulması için kullanılır. Bir yedek bozulursa, Mirror ile alınan diğer yedek kullanılabilir.

Veritabanı yedekleme işlemleri için gerekli söz dizimi oldukça farklı özellik ve seçeneklere sahiptir. İhtiyaçlara göre değişecek söz dizimi ile birçok farklı işlem kolaylıkla yapılabilir. Ancak hızlı yedekleme işlemlerinde Management Studio gibi yönetim araçları daha faydalı olacaktır.

Bir veritabanı yedeği dosyası hakkında bilgi almak gerekebilir. Yedek içerisinde bulunan, veritabanına ait dosya adları, uzantıları ve boyutu gibi bilgileri elde etmek için aşağıdaki sorgu kullanılabilir.

---

```
RESTORE FILELISTONLY
FROM DISK = 'C:\Backups\AWorks1.BAK';
```

---

	LogicalName	PhysicalName	Type	FileGroupName	Size	MaxSize	FileId	CreateLSN	DropLSN	UniqueId
1	AdventureWorks2012_Data	C:\Program Files\Microsoft SQL Server\MSSQL11.MSS... D	PRIMARY		214958080	35184372080640	1	0	0	40FA46CD-DFA9-40E1-90E5-5BE7CA6783EA
2	AdventureWorks2012_Log	C:\Program Files\Microsoft SQL Server\MSSQL11.MSS... L	LOG		1835008	35184372080640	2	0	0	E16FAF18-F2EA-4BF8-988A-7F30D4F87B84

Genel olarak kullanılabilecek ve birçok farklı parametreyi içeren bir yedekleme sorgusu oluşturalım. Yedekleme işlemleri genel olarak, bir script olarak hazırlanır ve sürekli kullanılır.

---

```
BACKUP DATABASE AdventureWorks
TO DISK = N'C:\Backups\AWorks10.BAK'
WITH NOFORMAT, NOINIT,
NAME = N'AdventureWorks - Full Database Backup',
SKIP, NOREWIND, NOUNLOAD, COMPRESSION,
STATS = 10, CHECKSUM, CONTINUE_AFTER_ERROR
GO

DECLARE @BackupSetID AS INT
SELECT @BackupSetID = position
FROM msdb..backupset
WHERE database_name = N'AdventureWorks'
AND
backup_set_id = (SELECT MAX(backup_set_id)
                  FROM msdb..backupset
                  WHERE database_name = N'AdventureWorks')
IF @BackupSetID IS NULL
BEGIN RAISERROR(N'Doğrulama başarısız! ''AdventureWorks2012'' bilgisi bulunamadı.', 16, 1)
END
RESTORE VERIFYONLY FROM DISK = N'C:\Backups\AWorks10.BAK'
WITH FILE = @BackupSetID,
NOUNLOAD,
NOREWIND
GO
```

---

Bu sorgu sonucunda, **Messages** alanındaki çıktı aşağıdaki gibi olacaktır.

```
10 percent processed.
20 percent processed.
30 percent processed.
40 percent processed.
50 percent processed.
60 percent processed.
70 percent processed.
80 percent processed.
90 percent processed.
Processed 24792 pages for database 'AdventureWorks', file 'AdventureWorks2012_Data' on file 4.
100 percent processed.
Processed 2 pages for database 'AdventureWorks', file 'AdventureWorks2012_Log' on file 4.
BACKUP DATABASE successfully processed 24794 pages in 3.716 seconds (52.126 MB/sec).
The backup set on file 4 is valid.
```

## T-SQL İLE TRANSACTION LOG DOSYASI YEDEĞİ OLUŞTURMAK

Gerçek uygulamalarda, veritabanı yedeğinin yanı sıra, transaction log dosyasını da yedeklemeniz gereken birçok senaryo ile karşılaşılır. Söz dizimi olarak veritabanına benzer yapıdadır.

**BACKUP LOG** ile transaction log dosya yedeği oluşturabilmek için en az bir tam veritabanı yedeği (*Full Backup*) alınmış olmalıdır. Aşağıdaki sorguları çalıştırmadan önce bir tam veritabanı yedeği alın. Aksi halde **BACKUP LOG** sorguları hata üretecektir.

**BACKUP LOG** ile oluşturulan transaction log dosyasının varsayılan uzantısı **TRN**'dir.

**AdventureWorks** veritabanının transaction log dosya yedeğini oluşturalım.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'C:\Backups\AWorks1.TRN';
```

---

Şifreli bir transaction log dosya yedeği oluşturalım.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'C:\Backups\AWorks2.TRN'
WITH PASSWORD = 'D!i@J#i$B#i$L';
```

---

Transaction log dosya yedeği oluşturulurken, yedek işleminin tamamlanma yüzdesini görelim.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'C:\Backups\AWorks3.TRN'
WITH STATS;
```

---

Yukarıdaki kullanım, varsayılan olarak %10 tamamlanma yüzdesi ile gerçekleşir.

%10 tamamlanma yüzdesini değiştirek %1 yapalım.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'C:\Backups\AWorks4.TRN'
WITH STATS = 1;
```

---

Transaction log dosyası yedeğine açıklama ekleyelim.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'C:\Backups\AWorks5.TRN'
WITH DESCRIPTION = 'AdventureWorks transaction log yedeği';
```

---

Yedekleme sırasında, birden fazla özellik tanımlayabilmek için şu ifade kullanılabilir. Bu sorguda, hem Mirror işlemi, hem de birden fazla özellik tanımlaması yapalım.

---

```
BACKUP LOG AdventureWorks
TO DISK = 'D:\Backups\AWorks6.TRN'
MIRROR TO DISK = 'D:\AdventureWorks_mirror.TRN'
WITH FORMAT;
```

---

Mirror, hazırlanan yedeğin bir kopyasının daha oluşturulması için kullanılır. Bir yedek bozulursa, Mirror ile alınan diğer yedek kullanılabilir.

## **SQL AGENT İLE OTOMATİK YEDEKLEME PLANI OLUŞTURMAK**

Veritabanı yedekleme işlemi, insan faktörüne bırakılmayacak kadar önemli ve düzenli gerçekleştirilmesi gereken bir görevdir. SQL Agent ile yedekleme işini, veritabanının otomatik olarak gerçekleştirmesi sağlanabilir.

SQL Agent, SQL Server'dan ayrı olarak çalışan özel bir servise sahiptir. Bu servisi başlatmak için aşağıdaki adımları izleyin.

- **Başlat** menüsünde, SQL Server araçları arasında **SQL Server Configuration Manager** aracını açın.
- **SQL Server Agent** servisi üzerine fare ile sağ tıklayın ve özelliklerine girin.
- **Log On tab** menüsü içerisinde, fare ile **Start** butonuna tıklayın.
- **Service tab** menüsü içerisinde, fare ile **Start Mode** olarak **Automatic** seçili hale getirin.
- İlk olarak **Uygula**, daha sonra **Tamam** butonuna fare ile tıklayarak işlemi tamamlayın.

**SQL Server Configuration Manager** içerisinde değerler aşağıdaki gibi olmalıdır.

 SQL Server Agent (MSSQLSERVER)	Running	Automatic
------------------------------------------------------------------------------------------------------------------	---------	-----------

Bu ayarları yaptıktan sonra, sistem yeniden başlasa bile, SQL Agent servisi sürekli ve otomatik olarak çalıştırılacaktır. Veritabanı yedekleme işlemini bu servisin otomatik olarak yapabilmesi için bu ayarın yapılmış olması gereklidir.

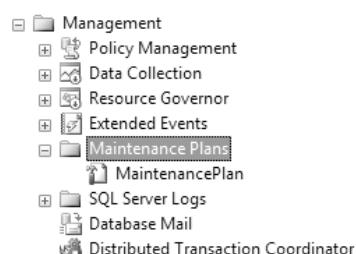
SQL Agent servisini açtıktan sonra, **Agent XPs** ayarlarının da açılması gereklidir. Açık değil ise, aşağıdaki sorgu ile bu ayarlar açılabilir.

---

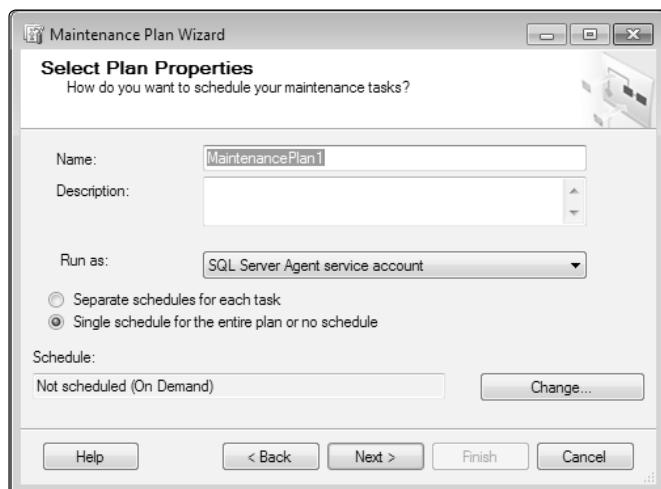
```
EXEC sys.sp_configure N'show advanced options',N'1';
RECONFIGURE WITH OVERRIDE;
EXEC sys.sp_configure N'Agent XPs', N'1';
RECONFIGURE WITH OVERRIDE;
EXEC sys.sp_configure N'show advanced options', N'0';
RECONFIGURE WITH OVERRIDE;
```

---

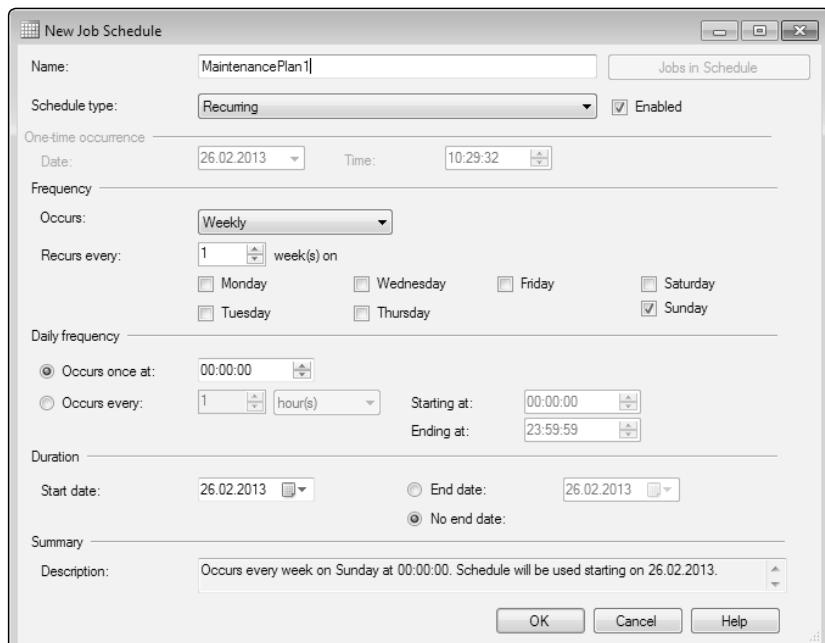
SQL Agent ile çalıştırılacak bir yedekleme planı oluşturmak için, **Management Studio**'da **Object Explorer** panelindeki **Management** klasörü içerisinde bulunan Maintenance Plans sekmesi fare ile sağ tıklanarak açılır. Açılan menüden, Maintenance Plan Wizard seçilerek yedekleme planı oluşturmaya başlanır.



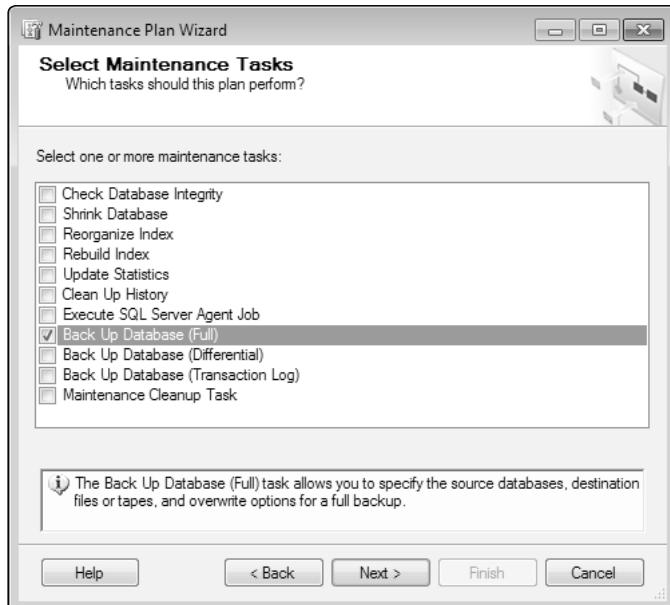
Açılan **Select Plan Properties** ekranını aşağıdaki gibi hazırlayın.



Ekranın Schedule kısmındaki **Change** butonuna tıklayarak, hangi zaman ve tarih aralığında yedekleme yapılacağını, yedeğin günlük, haftalık ya da aylık mı olduğunu ve hangi saatlerde yapılması gereği gibi detay bilgileri belirlenebilir.



Sonraki **Select Maintenance Tasks** ekranında **Back Up Database (Full)** seçim kutusu seçili hale getirilir.

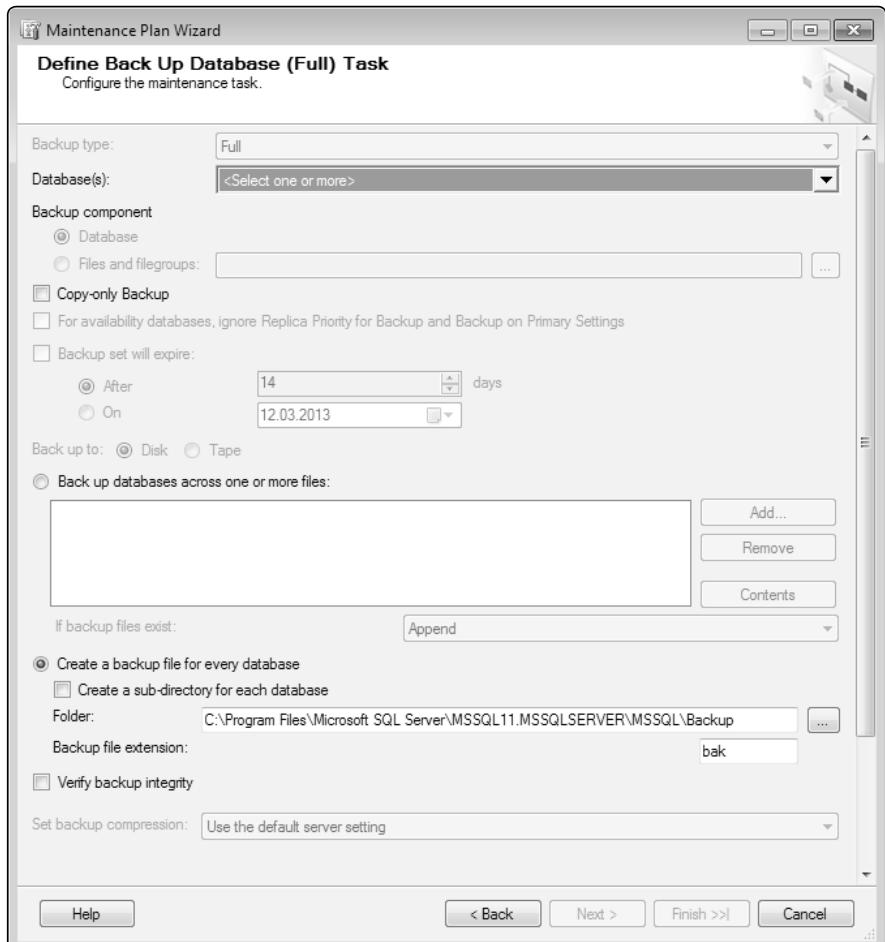


Sonraki **Select Maintenance Task Order** ekranını **Next** ile geçin.

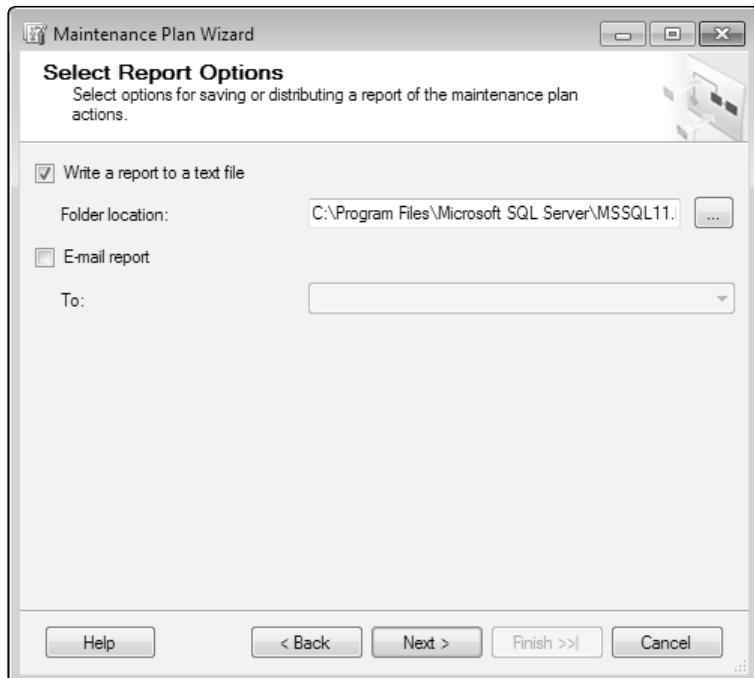


Sonraki, **Define Back Up Database (Full)** Task ekranındaki **Databases** seçim alanında **These databases**'i seçerek **AdventureWorks** veritabanını seçin ve **OK** düğmesine tıklayın.

Bu ekranda, yedeğin sıkıştırılma ayarı, dosya yolu, uzantısı, yedeğin diske mi teyp'e mi yapılacak gibi birçok ayar değiştirilebilir.



Son olarak, **Select Report Options** ekranında “*Write a report to a text file*” seçeneği seçilerek, yedekleme işlemi raporunun bir metin dosyasına yazılması sağlanabilir. Eğer gerekli ayarlar yapıldıysa, raporun e-mail ile gönderilmesi de sağlanabilir.



Tüm işlemler tamamlandıktan sonra **Complete the Wizard** ekranında, yedekleme planı ile ilgili özet ayarlar incelenebilir.

Plan oluşturulduktan sonra, **Object Explorer**daki **Maintenance Plans** içerisinde oluşturulan plan ismine sağ tıklayarak, **View History** menüsü seçildiğinde, yedekleme işleminin yapılmış yapılmadığı ve karşılaşılan sorunların takibi yapılabilir.

## VERİTABANINI GERİ YÜKLEMEK

Veritabanından alınan yedeklerin geri yüklemesi için **Management Studio** ve **T-SQL** olmak üzere iki yöntem kullanılabilir.

## T-SQL GERİ YÜKLEME KOMUTLARINI KULLANMAK

T-SQL ile geri yükleme komutlarını kullanabilmek için söz dizimi özelliklerini bilmeniz gereklidir. Temel anlamda geri yükleme komutları kullanımı basittir. Ancak, istek ve geri yükleme gereksinimleri bazen karmaşık sorgular oluşturmayı gerektirebilir.

- **FORMAT**: Yedek setinde bölme ve yedek dosyaları üzerinde yazma işlemini sağlar.
- **INIT ve NOINIT**: Yedek üzerine yazma ya da ekleme ayarları için kullanılır. SQL Server'da varsayılan özellik NOINIT'dir. NOINIT ile yedekleme, mevcut yedek dosyası ve setine ekleme olarak yapılır. INIT ile yedekleme, mevcut yedek verinin üzerine yazılmasını sağlar.
- **RESTART**: SQL Server'ın yedekleme işleminin kesildiği yerden devam etmesini sağlar.
- **UNLOAD**: Yedekleme işleminden sonra, teybi geriye sarip çıkarır. Varsayılan kullanımıdır.
- **NOUNLOAD**: Yedekleme işleminden sonra, teybi geriye sarmaz ve çıkarmaz. UNLOAD'ın tersidir.
- **BLOCKSIZEx**: Teyp üzerindeki fiziksel blok büyüklüğünü ayarlar.
- **SKIP**: Teyp üzerinde yer alan ANSI etiket bilgilerini atlar. Bu etiketler, teyp kullanım zamanının dolması, yazma gibi bilgileri içerir.
- **NOSKIP**: Varsayılan olarak SQL Server, ANSI teyp etiketlerini okur. Bu özellik, etiketlerin okunmaması istendiğinde kullanılır.

**AdventureWorks** veritabanını silelim ve yedekten geriye dönerek tekrar elde edelim. Bu işlemde önce, ilk olarak, **AdventureWorks** veritabanına ait bir tam veritabanı yedeğine (Full Database Backup) sahip olduğunuzu emin olun. Geri yükleme işlemi bu yedek ile gerçekleştirilecek.

Şimdi, **AdventureWorks** veritabanını silin.

---

```
DROP DATABASE AdventureWorks;
```

---

Veritabanı dosyalarının bulunduğu disk üzerindeki klasörde **AdventureWorks** veritabanına ait MDF ve LDF dosyalarının bulunmadığından emin olun. Dosyalar var ise, silin ya da farklı bir klasöre taşıyın.

```
RESTORE DATABASE AdventureWorks  
FROM DISK = N'C:\Backups\AWorks10.BAK';
```

---

Bu işleminden sonra **Management Studio**'daki **Object Explorer** panelinde, yeni bir veritabanının olduğu görülebilir.

# SQL SERVER MANAGEMENT OBJETS’I KULLANMAK

21

Bu bölüme kadar, SQL Server’ın T-SQL sorgu geliştirme yöntem ve yetenekleriyle, bazı veritabanı yöneticiliği konularını inceledik. Hem yazılım geliştirici, hem de veritabanı yöneticilerinin ortak ihtiyacı olan bir diğer özel konu ise, özel veritabanı işlem ve yönetimlerini gerçekleştirmek için, **Management Studio** gibi bir veritabanı yönetim ve geliştirme istemci aracı tasarlamak ve programsal olarak geliştirmektir.

Management Studio gibi bir istemci aracının olmadığı ortamlarda ya da yapılan bazı işlemlerin farklı bir kullanıcı arayüzü ile yönetilmek istendiğinde, .NET alt yapısını kullanarak veritabanına doğrudan ulaşılabilir. Veritabanı motoruna doğrudan bağlanarak, nesne oluşturma, veri yönetimi, sunucu yönetimi gibi veritabanı programlama tarafında gerçekleştirilecek birçok önemli işlemin yönetiminin yapılmasını sağlar. Bunları sağlayan nesne modelinin adı SMO, yani SQL Server Management Objects’dir.

Neler yapılabilir?

- Prosedür, trigger, view gibi nesneler oluşturmak.
- Veritabanı yedeği (*backup*) alma ve yedekten geri dönme işlemi gerçekleştirmek.
- Veritabanı script’i elde etmek.
- Bir veritabanı oluşturmak.
- Bir transaction oluşturmak.

- Bir job ve agent ile ilişkili diğer görevler yerine getirmek.
- LinkedServer oluşturmak.
- Index oluşturmak ve yönetmek.
- Kullanıcı tanımlı fonksiyon oluşturmak.
- SQL Server'da sunucusunda meydana gelen olayların yakalanması ve olay durumunu yönetmek.
- Sunucudaki nesne tiplerine koleksiyon olarak referans verme yeteneği. (T-SQL'de koleksiyon yoktur).
- Veritabanı şeması oluşturmak.
- SQL Server mail işlemlerini yönetmek.
- Veritabanı özellikleri ve ayarları hakkında bilgi almak.
- FullTextServices yönetimi.

Bu ve bunlar gibi birçok nesne SMO ile geliştirilebilir ve yönetilebilir.

## **SQL SERVER MANAGEMENT OBJECTS UYGULAMALARI**

SMO nesne modelini anlamanın en iyi yolu, birçok yeteneğini kullanarak, ihtiyaçlar doğrultusunda neler yapılabileceğini görmektir.

SQL Server'da, SMO ile uygulama geliştirebilmek için Visual Studio.NET editörü kullanılır. Bu kitaptaki örnek uygulamalarda, Visual Studio.NET editörü ve C# programlama dilini kullanacağız.

Bu bölüm, belirli seviyede programlama becerisi ve C# programlama dili bilgisi gerektirir. .NET tabanlı bir programlama dili kullanıyorsanız C# yerine, geliştiricisi olduğunuz programlama dilini de kullanabilirsiniz.

Programlama ortamında SMO nesne modelini kullanabilmek için bir Windows ya da konsol (Console) uygulaması oluşturun. Örneklerimizde Windows uygulaması geliştireceğiz. Ancak, bazı kodları değiştirerek konsol uygulaması ile daha hızlı sonuç elde edilebilir.

SMO nesnelerinin kullanılabilmesi için projeye aşağıdaki DLL'lerin referans olarak eklenmesi gereklidir.

---

```
Microsoft.SqlServer.ConnectionInfo
Microsoft.SqlServer.Management.Sdk.Sfc
Microsoft.SqlServer.Smo
Microsoft.SqlServer.SmoExtended
Microsoft.SqlServer.SqlEnum
Microsoft.SqlServer.SmoEnum
```

---

Belirtilen bu DLL'ler ile kitaptaki tüm örnekler gerçekleştirilebilir.

Eklenen DLL'lerin proje formlarında kullanılabilmesi için form içerisindeki, yukarıda bulunan `using` bölümünde bazı namespace tanımlamaları yapılması gerekecektir. Namespace tanımlamaları yapılmazsa ya da yukarıdaki DLL'ler eklenmezse SMO nesneleri aşağıdaki gibi altlarında kırmızı çizgi ve siyah yazı fontu ile görünecektir.

```
conn = new ServerConnection("dijibil-pc","sa","cihan..");
```

The type or namespace name 'ServerConnection' could not be found (are you missing a using directive or an assembly reference?)

Böyle bir görüntü oluştuğunda, projeye ilgili DLL'lerin referans edilip edilmediği kontrol edilmelidir. Sorun halen devam ediyorsa, formun `using` kısmındaki namespace tanımlamalarına bakılmalıdır.

Bir nesnenin namespace'ini kolay yoldan eklemenin iki yolu vardır.

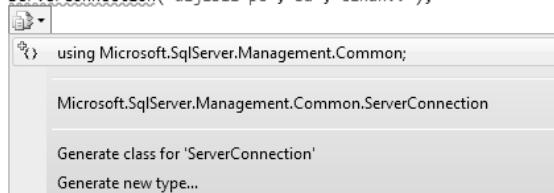
Klavye kısa yolları ile namespace eklemek;

- Altı kırmızı çizgili olan nesnenin üzerine fare ile tıklayın.
- **CTRL+ALT+F10** tuş kombinasyonunu kullanarak namespace ekleme penceresini açın ve ilgili namespace'i using kısmına ekleyin.

Fare ile namespace eklemek;

- Altı kırmızı çizgili olan nesnenin üzerine fare ile tıklayın.
- Nesne isminin sol alt köşesinde beliren mavi çizginin üzerine tıklayın ve açılan namespace ekleme penceresinde namespace eklemesini gerçekleştirin.

```
conn = new ServerConnection("dijibil-pc","sa","cihan..");
```





Kitabın bu bölümünde, SMO konusunu detaylarıyla kavrayabilmek için bir çok uygulama gerçekleştireceğiz. Bu uygulamaların kaynak kodlarını kitabıń CD ekinde bulabilirsiniz.

SMO ile ilgili hazırlanan uygulamaların ana ekran görüntüsü aşağıdaki gibidir.



Resimde görünen tüm örnekleri sırasıyla, parçalar halinde yapacağız.

## SMO İLE SUNUCU BAĞLANTISI OLUSTURMAK

SMO ile uygulama geliştirebilmek için her uygulamada gerekli olan ortak özellik, sunucuya bir bağlantı ile bağlı olmaktadır. Bu bağlantıyı oluşturabilmek için sunucu nesnesinin bir örneğine ve SQL Server erişim bilgilerine sahip olunmalıdır.

**Windows Authentication** ile sunucu bağlantısı gerçekleştirmek için aşağıdaki yöntem kullanılır.

---

```
ServerConnection conn = new ServerConnection();
conn.LoginSecure = true;
Server svr = new Server(conn);
svr.ConnectionContext.Connect();
```

---

**SQL Server Authentication** ile sunucu bağlantısı gerçekleştirmek için aşağıdaki yöntem kullanılır.

```
conn = new ServerConnection(
    ▲ 5 of 6 ▼ ServerConnection.ServerConnection(string serverInstance, string userName, string password)
    Initializes a new instance of the Microsoft.SqlServer.Management.Common.ServerConnection class with the specified server instance and logon credentials.
    serverInstance: A System.String value that specifies the name of the instance of the server with which the connection is established.
```

---

```
ServerConnection conn = new ServerConnection("dijibil-pc","sa","cihan..");
conn.LoginSecure = true;
Server svr = new Server(conn);
svr.ConnectionContext.Connect();
```

---

ADO.NET ya da farklı veri erişim katmanları ile veritabanı uygulamaları geliştirenler için bu kodun ne işe yaradığını anlamak zor değildir. Bu sorguda, sunucuya bağlantı oluşturabilmek için **ServerConnection** nesnesinin örneği oluşturuldu. Daha sonra bu bağlantı, sunucu işlemlerini yapmak için kullanılacak **Server** nesnesinin örneğine parametre olarak verildi.

Artık **conn** isimli bağlantı örneği kullanılarak, Server nesnesi yardımıyla sunucu iletişimini sağlanabilir.

## SUNUCU ÖZELLİKLERİNI ELDE ETMEK

Sunucu ile ilgili özellikler birçok programsal durumda gerekli olabilir. Sunucunun versiyon güncellemeleri, uyumluluk vb. bir çok durumda farklı özelliklerin elde edilmesi gereklidir. Bu işlem için SMO kullanılabilir.

Basit bir form oluşturup, içerisinde bir ListBox kontrolü yerleştirdikten sonra aşağıdaki kodları yazalım.

---

```
Server srv = new Server(conn);
listBox1.Items.Add("Sunucu: " + srv.Name);
listBox1.Items.Add("Versiyon: " + srv.Information.Edition);
listBox1.Items.Add("Veritabanı Sayısı : " + srv.Databases.Count);
listBox1.Items.Add("Instance Adı : " + srv.InstanceName);
listBox1.Items.Add("Net Adı : " + srv.NetName);
listBox1.Items.Add("Sunucu Tipi : " + srv.ServerType);
listBox1.Items.Add("Yayın : " + srv.Information.Edition);
listBox1.Items.Add("Dil : " + srv.Information.Language);
listBox1.Items.Add("İ.S Versiyon : " + srv.Information.OSVersion);
listBox1.Items.Add("Platform : " + srv.Information.Platform);
listBox1.Items.Add("Ürün Seviye : " + srv.Information.ProductLevel);
listBox1.Items.Add("Versiyon : " + srv.Information.Version + " "
+ srv.Information.VersionMajor + " "
+ srv.Information.VersionMinor);
listBox1.Items.Add("Yapı Numarası : " + srv.Information.BuildNumber);
listBox1.Items.Add("Collation : " + srv.Information.Collation);
```

---

`Server` nesnesinin örneği oluşturulurken `conn` parametresi gönderildiğine dikkat edin. Bu parametre, veritabanı bağlantısı oluşturmak için hazırladığımız bir `ServerConnection` nesnesi örneğidir. Sorgu yoğunluğu ve kitapta fazladan yer kaplamaması için bu ve bundan sonraki hiç bir SMO uygulamasında ek bir bağlantı tanımlama sorgusu yazmayacağız.

Proje çalıştırıldığında aşağıdaki gibi bir görüntü elde edilecektir.



Veritabanı ile ilgili sorguda istenen teknik özellikler listelenmektedir.

## VERİTABANLARI, DOSYA GRUPLARI VE DOSYALARIN LISTESİNİ ALMAK

SMO ile sunucudaki veritabanları, dosya grupları ve dosyalar ile ilgili bilgileri elde etmek mümkündür.

Form üzerine üç adet `ListBox` yerleştirdik. Kod sayfasındaki kodlar ise aşağıdaki gibi biridir.

---

```
Server srv = new Server(conn);
foreach (Database db in srv.Databases)
{
    listBox1.Items.Add(db.Name);
    foreach (FileGroup fg in db.FileGroups)
    {
        listBox2.Items.Add(" " + fg.Name);
        foreach (DataFile df in fg.Files)
        {
            listBox3.Items.Add(df.Name);
        }
    }
}
```

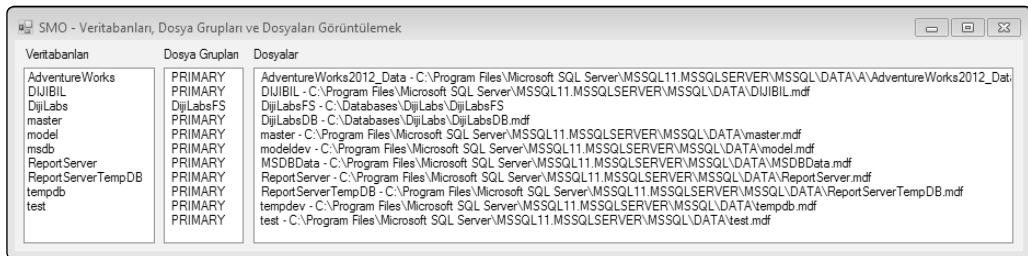
```

    {
        listBox3.Items.Add(" " + df.Name + " - " + df.FileName);
    }
}
}

```

---

Proje çalıştırıldığında aşağıdaki gibi görünecektir.



Solda bulunan **ListBox**'da sunucudaki veritabanları, ortadaki **ListBox**'da dosya grupları ve sağdaki **ListBox**'da ise veritabanı dosyalarının isimleriyle sistemdeki dosya yolları yer almaktadır.

## VERİTABANI ÖZELLİKLERİ ALMAK

Bir veritabanının özelliklerini ve bu özelliklerin değerlerini elde etmek için SMO kullanılabilir.

Form üzerine bir **ListBox** yerleştirerek formun **Load event**'ine aşağıdaki kodları yazalım.

---

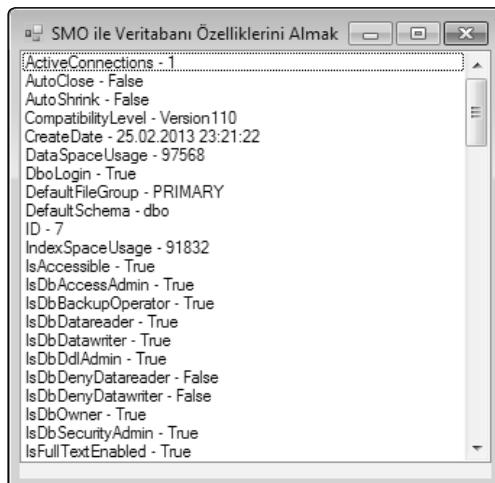
```

Server srv = new Server(conn);
Database database = srv.Databases["AdventureWorks"];
foreach (Property prop in database.Properties)
{
    listBox1.Items.Add(prop.Name + " - " + prop.Value);
}

```

---

Programın çalıştırıldıkten sonraki görünümü aşağıdaki gibi olacaktır.



## SUNUCUDAKİ TÜM VERİTABANLARINI LİSTELEMEK

Sunucuda bulunan tüm veritabanlarını listeleme işlemi aşağıdaki gibi yapılabilir.

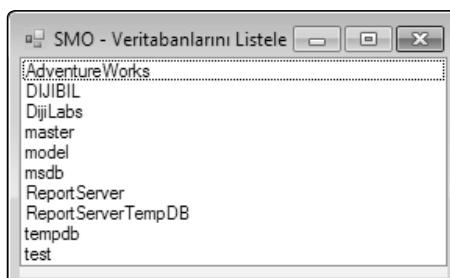
---

```
Server srv = new Server(conn);
srv.ConnectionContext.Connect();
Database db = srv.Databases["AdventureWorks"];

foreach (Database vt in srv.Databases)
{
    ListBox1.Items.Add(vt.Name.ToString());
}
```

---

Programın çalıştırıldıktan sonraki görünümü aşağıdaki gibi olacaktır.



# VERİTABANI OLUŞTURMAK

SMO ile yeni bir veritabanı oluşturulabilir.

Yeni bir veritabanı oluşturmak için kullanılacak bir metot geliştirelim.

---

```
void VeritabaniOlustur(string vt_ad)
{
    Server srv = new Server(conn);
    Database database = new Database(srv, "" + vt_ad + "");
    database.FileGroups.Add(new FileGroup(database, "PRIMARY"));
    DataFile dtPrimary = new DataFile(database.FileGroups["PRIMARY"],
        "Data", @"C:\Databases\"
        + vt_ad + ".mdf");
    dtPrimary.Size = 77.0 * 1024.0;
    dtPrimary.GrowthType = FileGrowthType.KB;
    dtPrimary.Growth = 1.0 * 1024.0;
    database.FileGroups["PRIMARY"].Files.Add(dtPrimary);

   LogFile logFile = new LogFile(database, "Log",
        @"C:\Databases\"
        + vt_ad + ".ldf");
    logFile.Size = 7.0 * 1024.0;
    logFile.GrowthType = FileGrowthType.Percent;
    logFile.Growth = 10.0;

    database.LogFiles.Add(logFile);
    database.Create();
    database.Refresh();
}
```

---



Yeni veritabanının GrowthType değerini belirtmek için kullanılan FileGrowthType enum değerinin hata üretmemesi için aşağıdaki DLL projeye eklenmelidir.

---

Microsoft.SqlServer.SqlEnum

---

Projeye DLL eklemek için aşağıdaki yolu izleyin.

- **Solution Explorer** panelindeki **References** kısmına sağ tıklayın.
- **Add Reference** butonunu tıklayın. Açılan formda **.NET** tab menüsünden ilgili **DLL** seçimini yapın, **OK** butonuna tıklayın.

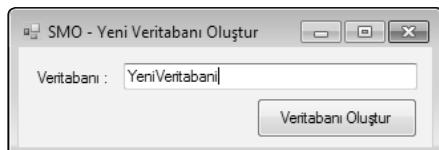
Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
VeritabaniOlustur(txtVeritabani.Text);
```

---

**KodLab** adında yeni bir veritabanı oluşturalım.



Programın çalışabilmesi için, bilgisayarınızın C:\ dizini içerisinde, **Databases** adında bir klasör bulunmalıdır.

İşlem herhangi bir hata üretmeden tamamlandıktan sonra, C:\ dizinindeki **Databases** klasörüne giderek, **KodLab** ismindeki **MDF** ve **LDF** dosyalarını görelim.

YeniVeritabani.ldf	25.02.2013 18:58	SQL Server Database Transaction Log File	7.168 KB
YeniVeritabani.mdf	25.02.2013 18:58	SQL Server Database Primary Data File	78.848 KB

**YeniVeritabani** isimli veritabanını **Management Studio** ile görüntüleyelim.



## VERİTABANI YEDEKLEMEK

SMO nesnelerinin yoğun kullanıldığı işlemlerden biri de veritabanı yedekleme işlemleridir. Veritabanı yedekleme işlemini gerçekleştirecek bir metot yazalım.

---

```
void Yedekle(string vt_ad)
{
    listBox1.Items.Clear();
    Random rnd = new Random();
    Server srv = new Server(conn);
    Database database = srv.Databases["" + vt_ad + ""];
    Backup backup = new Backup();
    backup.Action = BackupActionType.Database;
    backup.Database = database.Name;
    backup.Devices.AddDevice(@"C:\Backups\Backup"
        + rnd.Next(1, 10000)
```

```

        +".bak",
        DeviceType.File);
backup.PercentCompleteNotification = 10;
backup.PercentComplete += new PercentCompleteEventHandler(backup_
PercentComplete);
backup.SqlBackup(srv);
}

```

---

Veritabanı yedeklenirken, yedekleme işleminin % olarak tamamlanma oranı da öğrenilebilir. Bunun için, `backup` nesnesinin `PercentComplete` olayı kullanılmalıdır.

```

void backup_PercentComplete(object sender, PercentCompleteEventArgs e)
{
    listBox1.Items.Add("%" + e.Percent + " tamamlandı.");
}

```

---

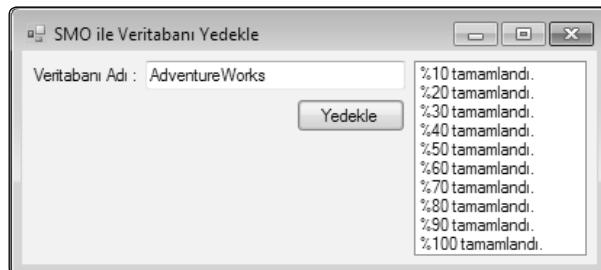
Yedekle isimli metot, bir butonun `click` olayı içerisinde şu şekilde kullanılabilir.

```
Yedekle(txtVeritabaniAd.Text);
```

---

Yedekleme işlemi için C:\ dizini içerisinde **Backups** klasörünü kullandık. Klasör içerisinde yedek isimleri **Backup** ile başlayacak ve devamında Random ile rastgele bir sayı oluşturularak, program kapatılmadan birden fazla yedekleme işlemi için farklı yedek dosyası isimleri oluşturulmasını sağladık.

**AdventureWorks** veritabanını yedekleyelim.



Yedeklenen veritabanı dosyası, C:\Backups klasöründe bulunmaktadır.

Backup4660.bak 25.02.2013 19:30 BAK Dosyası 198.772 KB

## VERİTABANI GERİ YÜKLEMİR

Yedeklenen veritabanını geri yükleme işlemi de, yedekleme işlemine benzer şekilde SMO ile programlanabilir.

Yedekten geri dönme işlemini gerçekleştirecek bir metot geliştirelim. Bu metot, ekranda bulunan **ProgressBar** ve **ListBox** kontrolleriyle, yükleme işleminin yüzde olarak tamamlanma oranını gösterecektir.

---

```
void VeritabaniGeriYukle(string vt_ad, string yedek_ad)
{
    Server srv = new Server(conn);
    Restore restore = new Restore();
    string fileName = @"C:\Backups\" + yedek_ad + ".bak";
    string databaseName = "" + vt_ad + "";
    restore.Database = databaseName;
    restore.Action = RestoreActionType.Database;
    restore.Devices.AddDevice(fileName, DeviceType.File);
    this.progressBar1.Value = 0;
    this.progressBar1.Maximum = 100;
    this.progressBar1.Value = 10;
    restore.PercentCompleteNotification = 10;
    restore.ReplaceDatabase = true;
    restore.PercentComplete += new
    PercentCompleteEventHandler(res_PercentComplete);
    restore.SqlRestore(srv);
}
```

---

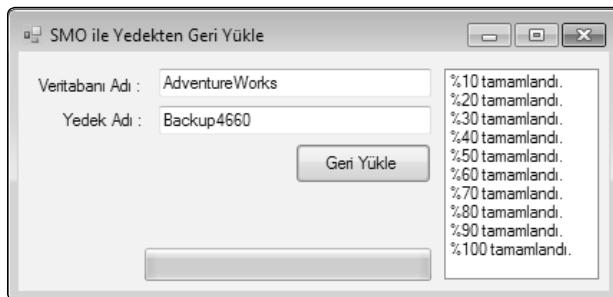
Yedeğin geri yüklenme işleminin % olarak tamamlanma oranını bir **ListBox**'a yazdıralım.

---

```
void restore_PercentComplete(object sender, PercentCompleteEventArgs e)
{
    progressBar1.Value = e.Percent;
    listBox1.Items.Add("%" + e.Percent + " tamamlandı.");
}
```

---

Program arayüzü aşağıdaki gibi görülmektedir.



Bu örnekte **C:\** dizinindeki **Backups** klasöründe bulunan **Backup4660.bak** isimli veritabanı yedek dosyasını SQL Server'a geri yükleyerek, **AdventureWorks** isimli yeni bir veritabanı oluşturduk.

## VERİTABANINI SİLMEK

SMO ile mevcut veritabanlarını silmek mümkündür.

Veritabanı silmek için, SMO nesneleriyle bir metod geliştirelim.

---

```
void VeritabaniSil(string vt_ad)
{
    Server srv = new Server(conn);
    Database db = srv.Databases["" + vt_ad + ""];
    db.Drop();
}
```

---

Metod, program içerisinde şu şekilde çağrılabılır.

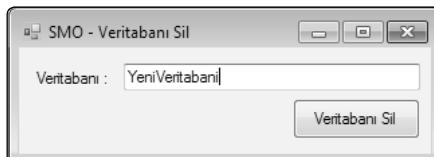
---

```
VeritabaniSil(txtVeritabani.Txt);
```

---

Veritabanı silmek kritik bir işlem olduğu için, SMO ile geliştirilen programlarda bazı kontrol ve kullanıcı onaylarına tabi tutulmalıdır. Örneğin; veritabanını silme butonuna basıldığında ekrana bir onay kutusu getirerek, kullanıcının bu işlemi gerçekten yapmak isteyip istemediği sorularak, ek bir onay alınabilir.

Daha önce oluşturduğumuz **YeniVeritabani** isimli veritabanını silelim.



## TABLO VE SÜTUNLARI LİSTELEMEK

SMO kullanarak, bir veritabanındaki tablo ve her tablo içerisindeki sütunları hiyerarşik bir düzen içerisinde listelemek mümkündür.

Bu işlemi **AdventureWorks** veritabanı için yapalım. **AdventureWorks** veritabanındaki tüm tablo ve tabloların içerisindeki sütunların listesini getirecek **Getir()** adında bir metot oluşturalım.

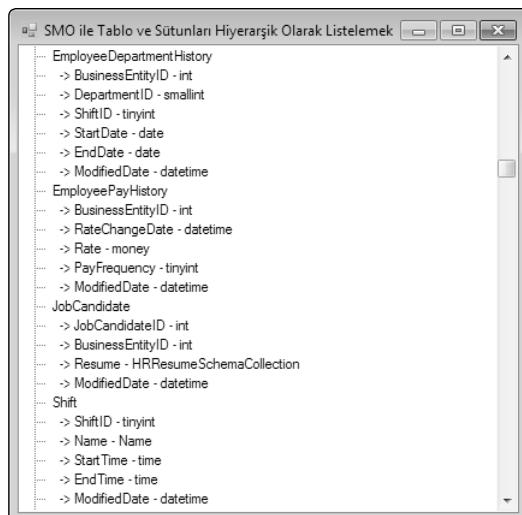
---

```
void Getir()
{
    Server srv = new Server(conn);
    Database db = srv.Databases["AdventureWorks"];

    foreach (Table table in db.Tables)
    {
        treeView1.Nodes.Add(" " + table.Name);
        foreach (Column col in table.Columns)
        {
            treeView1.Nodes.Add(" -> " + col.Name + " - "
                + col.DataType.Name);
        }
    }
}
```

---

Formun **Load** olayında **Getir()** metodunu çağrıdığımızda aşağıdaki gibi, **TreeView** kontrolü içerisinde hiyerarşik bir liste döndürecektr.



## VIEW OLUŞTURMAK

SMO nesne modeliyle view nesnesi programlanabilir.

View nesnesi oluşturmak için bir metot geliştirelim.

---

```
void ViewOlustur(string vt_ad, string view_ad,
                  string view_header, string view_body)
{
    try
    {
        Server srv = new Server(conn);
        Database database = srv.Databases["" + vt_ad + ""];
        Microsoft.SqlServer.Management.Smo.View myview = new
        Microsoft.SqlServer.Management.Smo.View(database, "" + view_ad + "");
        myview.TextHeader = "" + view_header + "";
        myview.TextBody = "" + view_body + "";
        myview.Create();
        MessageBox.Show(view_ad + " adındaki view başarıyla oluşturuldu");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

---

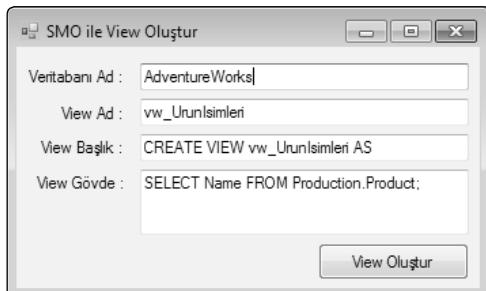
**ViewOlustur** isimli bu metot, bir butonun **Click** olayında aşağıdaki gibi çağrırlabilir.

---

```
ViewOlustur(txtVeritabaniAd.Text, txtViewAd.Text,
            txtViewHeader.Text, txtViewBody.Text);
```

---

Programı test etmek için **vw\_Urunler** isimli bir view oluşturalım.



**vw\_Urunler** isimli view'i SSMS içerisinde görüntüleyip sorgulayalım.

dbo.vw\_UrunIsimleri

---

```
SELECT * FROM vw_Urunler;
```

---

## STORED PROCEDURE OLUSTURMAK

SMO nesneleri kullanılarak Stored Procedure geliştirilebilir.

Programsal olarak prosedür oluşturmak için **ProsedurOlustur** isminde bir metot programlayalım.

---

```
void ProsedurOlustur(string vt_ad, string pr_ad, string pr_govde,
string param_ad)
{
    try
    {
        Server srv = new Server(conn);
        Database database = srv.Databases["" + vt_ad + ""];
        StoredProcedure sp = new StoredProcedure(database, "" + pr_ad + "");
        sp.TextMode = false;
        sp.AnsiNullsStatus = false;
        sp.QuotedIdentifierStatus = false;
        StoredProcedureParameter param = new StoredProcedureParameter
            (sp, "@" + param_ad + "", DataType.Int);
        sp.Parameters.Add(param);
        string spBody = "" + pr_govde + " = @" + param_ad + "";
        sp.TextBody = spBody;
        sp.Create();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Hata : " + ex.Message);
    }
}
```

---

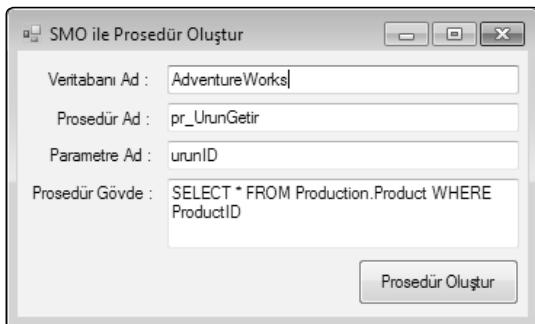
Bu metot, bir butonun **Click** olayında aşağıdaki gibi kullanılabilir.

---

```
ProsedurOlustur(txtVeritabaniAd.Text, txtProsedurAd.Text,
txtProsedurGovde.Text, txtParametreAd.Text);
```

---

**ProductID** bilgisine göre ürün bulup getiren bir prosedür oluşturalım.



**pr\_UrunGetir** prosedürüni SSMS ile SQL Server içerisinde test edelim.

---

```
pr_UrunGetir 1;
```

---

	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

## BİR STORED PROCEDURE'Ü OLUŞTURMAK, DEĞİŞİSTİRMEK VE SİLMEK

Bir prosedürü tek bir transaction içerisinde oluşturma, değiştirme ve silme işlemini programlanabilir.

Bu işlemi gerçekleştirmek için gerekli metodu geliştirelim.

---

```
void ProsedurDegistir(string vt_ad, string prosedur_ad,
                      string prosedur_govde, string param_ad1,
                      string param_ad2)
{
    Server srv = new Server(conn);
    srv.ConnectionContext.Connect();
    Database db = srv.Databases["" + vt_ad + ""];
    StoredProcedure sp = new StoredProcedure(db, "" + prosedur_ad + "");
    sp.TextMode = false;
    sp.AnsiNullsStatus = false;
    sp.QuotedIdentifierStatus = false;
    StoredProcedureParameter param = new StoredProcedureParameter(sp,
        "@" + param_ad1 + "", DataType.Int);
```

```

    sp.Parameters.Add(param);

    StoredProcedureParameter param2 = new StoredProcedureParameter(sp,
        "@param_ad2 + "", DataType.NVarChar(50));
    param2.IsOutputParameter = true;
    sp.Parameters.Add(param2);

    string sql = prosedur_govde;
    sp.TextBody = sql;

    sp.Create();
    sp.QuotedIdentifierStatus = true;
    listBox1.Items.Add("Prosedür oluşturuldu.");

    sp.Alter();
    listBox1.Items.Add("Prosedür değiştirildi.");
    sp.Drop();
    listBox1.Items.Add("Prosedür silindi.");
}

```

---

Metod, program içerisinde aşağıdaki şekilde kullanılabilir.

---

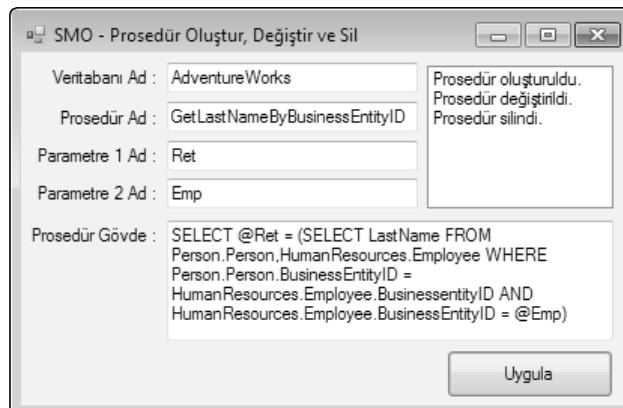
```

ProsedurDegistir(txtVeritabaniAd.Text,
    txtProsedurAd.Text,
    txtProsedurGovde.Text,
    txtParametre1Ad.Text,
    txtParametre2Ad.Text);

```

---

Program çalıştırıldıktan sonraki görünümü aşağıdaki gibi olacaktır.



## VERİTABANINDAKİ TÜM STORED PROCEDURE'LERİ ŞİFRELEMEK

Veritabanı yöneticisinin görevlerinden birisi de, kaynak kod güvenliğini sağlamaktır. Kitabın **Stored Procedure**'ler isimli bölümünde tüm detaylarıyla incelediğimiz güvenlik ve şifreleme işlemlerini SMO ile programsal olarak da yapmak mümkündür.

Bir veritabanındaki tüm prosedürleri tek seferde şifrelemek ya da tek bir prosedürü şifrelemek için SMO kullanılabilir.

Belirtilen veritabanındaki tüm prosedürleri şifreleyecek bir metot geliştirelim.

---

```
void ProsedurleriSifrele(string vt_ad)
{
    string dbRequested = vt_ad;

    var srv = new Server();
    try
    {
        srv = new Server(conn);
    }
    catch(Exception ex)
    {
        MessageBox.Show("Hata : " + ex.Message);
        Environment.Exit(Environment.ExitCode);
    }

    var db = new Database();
    try
    {
        db = srv.Databases[dbRequested];
        if (db == null)
            throw new Exception();
    }
    catch(Exception ex)
    {
        MessageBox.Show("Hata : " + ex.Message);
        Environment.Exit(Environment.ExitCode);
    }
}
```

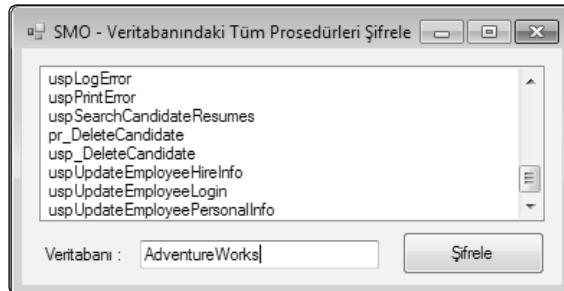
```
var sp = new StoredProcedure();
for (int i = 0; i < db.StoredProcedures.Count; i++)
{
    sp = db.StoredProcedures[i];
    if (!sp.IsSystemObject)
    {
        if (!sp.IsEncrypted)
        {
            sp.TextMode = false;
            sp.IsEncrypted = true;
            sp.TextMode = true;
            sp.Alter();
            listBox1.Items.Add(" " + sp.Name + Environment.NewLine);
        }
    }
}
```

Bu metodu program içerisinde kullanmak için, aşağıdaki gibi sadece veritabanı adını vermek yeterlidir.

Prosedurleri Sifrele ("AdventureWorks");

Metot, yukarıdaki gibi çalıştırıldığında, AdventureWorks veritabanında bu metot ile şifrelenen tüm prosedürlerin isimlerini geri döndürecektr.

Programı, üzerinde bir çok prosedür oluşturarak değiştirdiğim AdventureWorks veritabanı ile test ettiğimde aşağıdaki gibi görülmektedir.



## ŞEMA OLUŞTURMAK

SQL Server'da veritabanı yönetiminin önemli yardımcılarından birisi şemalardır. Veritabanındaki nesneleri şemalar içersine alarak mantıksal nesne yönetimi gerçekleştirilebilir. SMO ile de şema oluşturmak mümkündür.

Veritabanında yeni bir şema oluşturacak metot geliştirelim.

---

```

void SemaOlustur(string vt_ad, string sema_ad, string tablo_ad,
                  string id_sutun, string sutun_ad1)
{
    try
    {
        Server srv = new Server(conn);
        Database database = srv.Databases["" + vt_ad + ""];
        Schema schema = new Schema(database, "" + sema_ad + "");
        schema.Owner = "dbo";
        schema.Create();

        Table Kullanicilar = new Table(database, "" + tablo_ad + "", ""
+ sema_ad + "");

        DataType dt = new DataType(SqlDataType.Int);
        Column IDSutunu = new Column(Kullanicilar, "" + id_sutun + "", dt);
        IDSutunu.Nullable = false;
        IDSutunu.Identity = true;
        IDSutunu.IdentityIncrement = 1;
        IDSutunu.IdentitySeed = 1;
        Kullanicilar.Columns.Add(IDSutunu);
        dt = new DataType(SqlDataType.VarChar, 50);
        Column AdSutunu = new Column(Kullanicilar, "" + sutun_ad1 + "", dt);
        Kullanicilar.Columns.Add(AdSutunu);
        Kullanicilar.Create();
        MessageBox.Show("Şema ve tablo başarıyla oluşturuldu.");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Hata : " + ex.Message);
    }
}

```

---

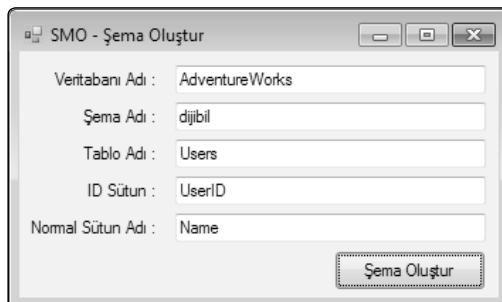
Bu metot program içerisinde aşağıdaki gibi kullanılabilir.

---

```
SemaOlustur(txtVeritabaniAd.Text,
            txtSemaAd.Text,
            txtTabloAd.Text,
            txtIDSutun.Text,
            txtNormalSutunAd.Text);
```

---

Programın örnek kullanımı aşağıdaki gibidir.



Şema oluşturulduktan sonra, **dijibil** şeması ile oluşturulan **Users** isimli tablo, **Object Explorer** panelinin **AdventureWorks** tabloları içerisinde görülebilir.

<input type="checkbox"/>	<b>dijibil.Users</b>
<input type="checkbox"/>	<b>Columns</b>
<input type="checkbox"/>	<b>UserID</b> (int, not null)
<input type="checkbox"/>	<b>Name</b> (varchar(50), null)

## ŞEMALARI LISTELEMEK

SMO ile bir veritabanındaki tüm şemalar listelenebilir.

Veritabanındaki tüm şemaları listeleyecek bir metot geliştirelim.

---

```
void SemalariListele()
{
    Server srv = new Server(conn);
    Database database = srv.Databases["AdventureWorks2012"];
    foreach (Schema schema in database.Schemas)
    {
        lstSemalar.Items.Add(schema.Name);
    }
}
```

---

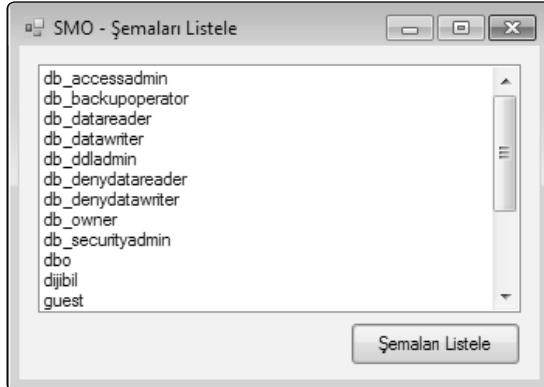
Metodun program içerisinde kullanımı aşağıdaki gibi basittir.

---

```
Semalarilistele();
```

---

Program çalıştırıldıkten sonra aşağıdaki gibi tüm şemaları listeleyecektir.



## LINKED SERVER OLUŞTURMAK

Bağlı sunucular ile çalışmak için SMO ile bir **LinkedServer** programlanabilir.

---

```
Server srv = new Server(@"Instance_A");
LinkedServer lsrv = new LinkedServer(srv, "Instance_B");
LinkedServerLogin login = new LinkedServerLogin();
login.Parent = lsrv;
login.Name = "Login_Instance_A";
login.RemoteUser = "Login_Instance_B";
login.SetRemotePassword("sifre");
login.Impersonate = false;
lsrv.ProductName = "SQL Server";
lsrv.Create();
login.Create();
```

---

## OTURUM OLUŞTURMAK

SMO ile yeni bir oturum (*login*) oluşturulabilir.

Yeni oturum oluşturmak için gerekli metodu geliştirelim.

---

```
void LoginOlustur(string login_ad, string sifre, string rol_ad)
{
    Server srv = new Server(conn);
    Login login = new Login(srv, "" + login_ad + "");
    login.LoginType = LoginType.SqlLogin;
    login.Create("+" + sifre + "+");
    login.AddToRole("+" + rol_ad + "+");
}
```

---

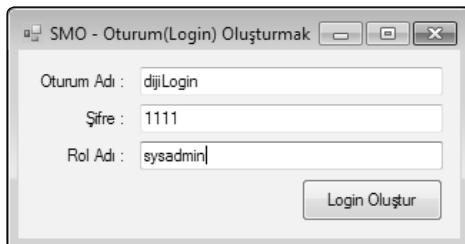
Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
LoginOlustur(txtOturumAd.Text, txtSifre.Text, txtRol.Text);
```

---

**dijiLogin** adında, **sysadmin** rolüne sahip yeni bir oturum oluşturalım.



**SQL Server**'da **Logins** bölümü içerisinde **dijiLogin** oturumunu **dijiLogin** görüntüleyelim.

**Login** üzerine iki kez tıklayarak özelliklerine girilip, **Server Roles** sekmesinden oturumun **sysadmin** rolüne sahip olduğu görülebilir.

## O TURUMLARI LİSTELEMEK

SMO ile SQL Server'daki oturumlar listelenebilir.

SQL Server'daki oturumları ve ilişkili olan veritabanları ile kullanıcı adlarının tamamını hiyerarşik olarak listeleyeceğim metodu geliştirelim.

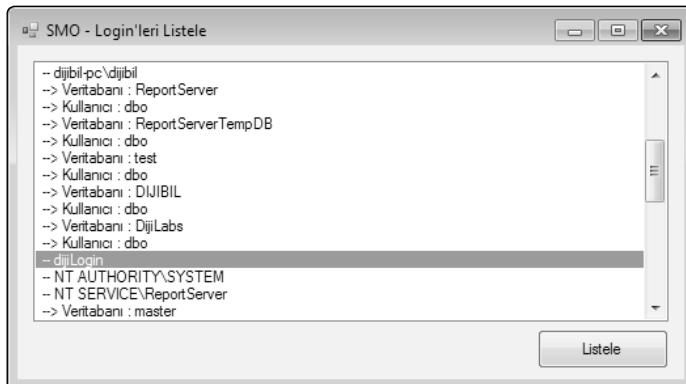
---

```
void LoginleriListele()
{
    Server srv = new Server(conn);
    foreach (Login login in srv.Logins)
    {
```

```
lstLoginler.Items.Add(" -- " + login.Name);
if (login.EnumDatabaseMappings() != null)
{
foreach (DatabaseMapping map in login.EnumDatabaseMappings())
{
    lstLoginler.Items.Add(" --> Veritabani : " + map.DBName);
    lstLoginler.Items.Add(" --> Kullanici : " + map.UserName);
}
}
}
```

Metot aşağıdaki gibi çağrılabılır.

```
LoginleriListele();
```



## KULLANICI OLUŞTURMAK

Bir veritabanında kullanıcı oluşturmak için SMO nesne modeli kullanılabilir.

Kullanıcı oluşturmak için gerekli metodu geliştirelim.

```
void KullaniciOlustur(string vt_ad, string kullanici_ad, string login_ad)
{
    Server srv = new Server(conn);
    Database db = srv.Databases["" + vt_ad + ""];
    User u = new User(db, "" + kullanici_ad + "");
    u.Login = "" + login_ad + "";
    u.Create();
}
```

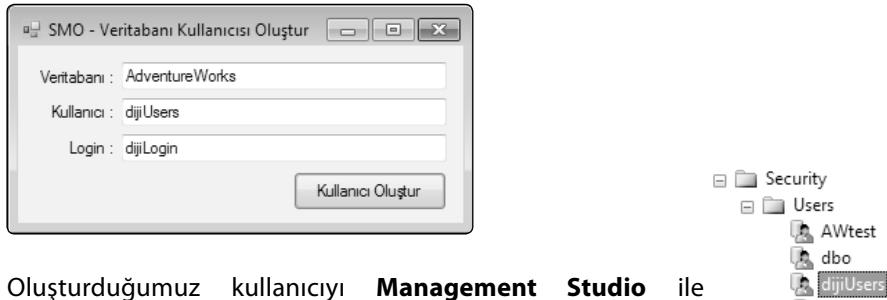
Metod, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
KullaniciOlustur(txtVeritabani.Text, txtKullanici.Text, txtLogin.Text);
```

---

**AdventureWorks** veritabanında yeni bir kullanıcı oluşturalım.



Oluşturduğumuz kullanıcıyı **Management Studio** ile görüntüleyelim.

## KULLANICILARI LISTELEMEK

SMO ile bir veritabanına ait kullanıcıların bilgileri elde edilebilir.

Kullanıcı bilgilerini elde etmek için gerekli metodu geliştirelim.

---

```
Server srv = new Server(conn);
Database db = srv.Databases["" + vt_ad + ""];
foreach (User user in db.Users)
{
    lstKullanicilar.Items.Add("Kullanıcı ID : " + user.ID);
    lstKullanicilar.Items.Add("Kullanıcı : " + user.Name);
    lstKullanicilar.Items.Add("Oturum : " + user.Login);
    lstKullanicilar.Items.Add("Varsayılan Şema : " + user.DefaultSchema);
    lstKullanicilar.Items.Add("Oluşturulma Tarih : " + user.CreateDate);
    lstKullanicilar.Items.Add("-----");
}
```

---

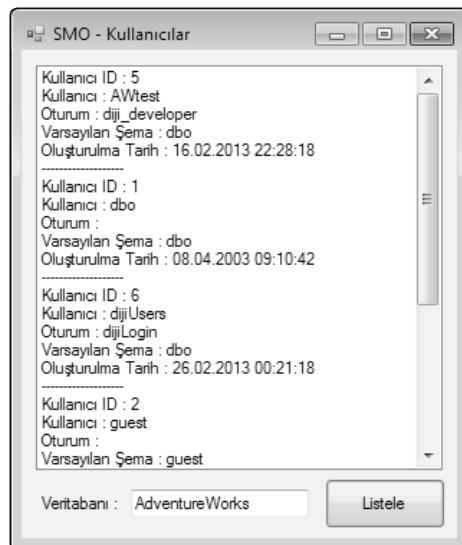
Metot program içerisinde aşağıdaki gibi çağrılabılır.

---

```
KullaniciListele(txtVeritabani.Text);
```

---

**AdventureWorks** veritabanındaki tüm kullanıcıların bilgilerini listeleyelim.



## ROL OLUSTURMAK

SMO ile veritabanında bir rol programlanabilir.

Rol oluşturmak için gerekli metodu geliştirelim.

---

```
void RolOlustur(string vt_ad, string rol_ad)
{
    Server srv = new Server(conn);
    Database db = srv.Databases["" + vt_ad + ""];
    DatabaseRole dbRole = new DatabaseRole(db, "" + rol_ad + "");
    dbRole.Create();
}
```

---

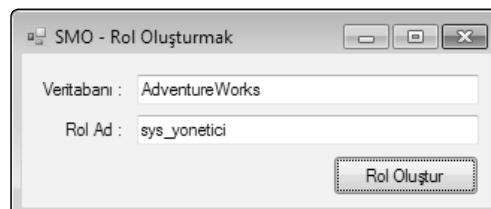
Metod, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
RolOlustur(txtVeritabani.Text, txtRolAd.Text);
```

---

**AdventureWorks** veritabanında, **sys\_yonetici** adında yeni bir rol oluşturalım.



Oluşturduğumuz rolü **Mangement Studio** ile görüntüleyelim.

- db\_securityadmin
- public
- sys\_yonetici

## ROLLERİ LISTELEMEK

SMO ile veritabanındaki rolleri listelenebilir.

Belirtilen veritabanına ait rolleri ve role bağlı üyeleri listelemek için bir metot geliştirelim.

---

```
void RolleriListele(string vt_ad)
{
    lstRoller.Items.Add("Ad : " + dr.Name);
    lstRoller.Items.Add("Oluşturma Tarihi : " + dr.CreateDate);
    lstRoller.Items.Add("Sahibi : " + dr.Owner);
    lstRoller.Items.Add("Rol Üyeleri :");
    foreach (string s in dr.EnumMembers())
        lstRoller.Items.Add(" " + s);
    lstRoller.Items.Add("-----");
}
```

---

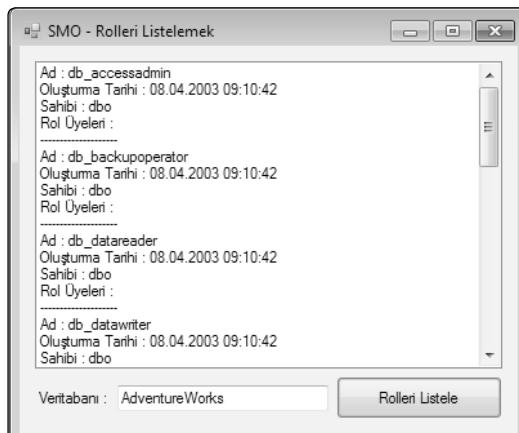
Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
RolleriListele(txtVeritabani.Text);
```

---

**AdventureWorks** veritabanındaki rollerin bilgilerini ve role bağlı üyeleri listeleyelim.



## ROL ATAMAK

SMO ile bir kullanıcıya rol ataması yapılabilir.

Bir rol ataması gerçekleştirecek metot geliştirelim.

---

```
void KullaniciRolAtama(string vt_ad, string kullanici_ad, string rol)
{
    Server srv = new Server(conn);
    Database db = srv.Databases["" + vt_ad + ""];
    User u = db.Users["" + kullanici_ad + ""];
    u.AddToRole("" + rol + "");
    u.Alter();
}
```

---

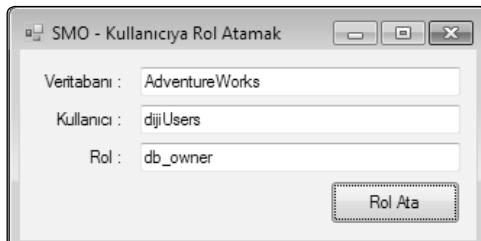
Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
KullaniciRolAtama(txtVeritabani.Text, txtKullanici.Text, txtRol.Text);
```

---

Programı kullanarak bir rol ataması gerçekleştirelim.



**Object Explorer** panelinden aşağıdaki yol takip edilerek eklenen rol görülebilir.

- AdventureWorks veritabanı içerisinde **Security** klasörünü açın.
- Security klasöründe **Users** klasörünü açın.
- dijiUsers** isimli kullanıcının üzerine iki kez tıklayarak, **Özellikler** penceresini açın.
- Açılan pencerede Membership bölümüne girin.

<input type="checkbox"/>	db_accessadmin
<input type="checkbox"/>	db_backupoperator
<input type="checkbox"/>	db_datareader
<input type="checkbox"/>	db_datawriter
<input type="checkbox"/>	db_ddladmin
<input type="checkbox"/>	db_denydatareader
<input type="checkbox"/>	db_denydatawriter
<input checked="" type="checkbox"/>	db_owner
<input type="checkbox"/>	db_securityadmin
<input type="checkbox"/>	sys_yonetici

## ASSEMBLY'LERİ LISTELEMEK

Veritabanındaki tüm Assembly'leri listelemek için SMO nesne modeli kullanılabilir.

Belirtilen veritabanına ait Assembly'leri listelemek için bir metod geliştirelim.

---

```
void Assemblyler(string vt_ad)
{
    Server srv = new Server(conn);
    Database db = srv.Databases["" + vt_ad + ""];

    foreach (SqlAssembly assembly in db.Assemblies)
    {
        lstAssemblyler.Items.Add("Assembly Adı : " + " " + assembly.Name);
        foreach (SqlAssemblyFile assemblyFile in assembly.SqlAssemblyFiles)
            lstAssemblyler.Items.Add(" " + assemblyFile.Name);
    }
}
```

---

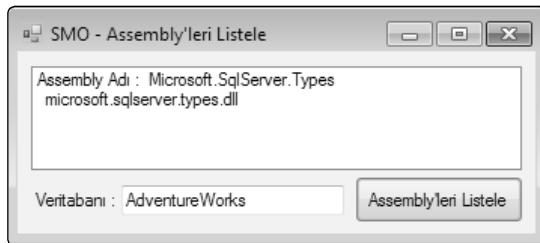
Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
Assemblyler(txtVeritabani.Text);
```

---

**AdventureWorks** veritabanındaki Assembly'leri listeleyelim.



## BİR TABLONUN SCRIPT'İNİ OLUŞTURMAK

Genellikle veritabanı yöneticisinin görevi olan, veritabanı tablolarının SQL script kodlarını saklama ihtiyacı, bazen yazılım geliştiriciler için de gerekli olur. Belirli bir şema içerisindeki belirtilen bir tablonun, SQL script kodlarını elde etmek için SMO nesne modeli kullanılabilir.

Bir tablonun SQL script kodlarını elde edecek metot geliştirelim.

---

```
string ScriptOlustur(string vt_ad, string tablo_ad, string sema_ad)
{
    Server srv = new Server(conn);
    srv.ConnectionContext.Connect();

    Database db = srv.Databases["" + vt_ad + ""];
    Table tablo = db.Tables["" + tablo_ad + "", "" + sema_ad + ""];
    StringCollection script = tablo.Script();
    string sql_script = string.Empty;

    foreach (string s in script)
    {
        sql_script = sql_script + s;
    }
    return sql_script;
}
```

---

Metot, program içerisinde aşağıdaki gibi çağrılabılır.

---

```
ScriptOlustur(txtVeritabani.Text, txtTablo.Text, txtSema.Text);
```

---

Dikkat ederseniz, **ScriptOlustur** metodu string geri dönüş tipine sahiptir. Metottan geri dönen değer, parametre ile belirtilen tablonun SQL script kodları olacaktır. Biz de bu geri dönen SQL kodlarını bir **TextBox'a** aktararak ekranda görüntüledik.

