

İÇİNDEKİLER

VERİTABANINA GİRİŞ	4
VERİ (DATA) NEDİR.....	4
VERİ TABANI (DATABASE) NEDİR.....	4
VERİTABANI SUNUCULARI (DATABASE SERVER)	4
MS-SQL SERVER (MICROSOFT SQL SERVER)	4
MS-SQL SERVER SÜRÜMLERİ.....	4
NORMALİZASYON.....	5
1NF	5
2NF	5
3NF	5
SQL SERVER MANAGEMENT STUDIO	6
SERVER TYPE.....	6
SERVER NAME.....	6
AUTHENTICATION	6
Sql Server Authentication	6
Windows Authentication.....	6
OBJECT EXPLORER	7
NEW DATABASE	7
NEW TABLE.....	8
CONSTRAINTS	8
SQL SERVER VERİ TİPLERİ	8
Tam sayı veri tipleri.....	8
Ondalıklı sayı veri tipleri	9
Tarih ve Saat veri tipleri.....	9
Metinsel veri tipleri.....	9
TRANSACT SQL (T-SQL)	9
SQL (Structured Query Language).....	9
T-SQL	9
T-SQL İLE KISITLAYICILARI KULLANARAK VERİ BÜTÜNLÜĞÜNÜ SAĞLAMA	9
TABLO OLUŞTURMA.....	10
PRIMARY KEY CONSTRAINT	10
UNIQUE CONSTRAINTS.....	11
FOREIGN KEY CONSTRAINT	11
CHECK CONSTRAINTS	12
DEFAULT CONSTRAINTS.....	12
TABLOYA VERİ EKLEME (INSERT)	12
T-SQL SORGULAMA ÖRNEKLERİ	13
SELECT KOMUTU	13
ALİASES (TAKMA AD)	14
WHERE KOMUTU	15
WHERE KOMUTU OPERATÖRLERİ	15
Karşılaştırma Operatörleri.....	15
Mantıksal Operatörler.....	15
Diğer Operatörler	15

In Operatörü	16
Not In Operatörü	17
LIKE OPERATÖRÜ	17
Order By (Sıralama)	17
Distinct (farklı kayıtları elde etme).....	18
Top	18
AGGREGATE FUNCTİONS	19
CASE WHEN	19
GROUP BY (GRUPLAMA)	20
GROUP BY VE COUNT FONKSİYONU.....	20
Group By ve Sum Fonksiyonu	21
Group by ve Having	22
INSERT, UPDATE, DELETE İŞLEMLERİ	22
Insert Komutu	22
Identity Column	23
Delete Komutu	24
Update Komutu	25
JOINS.....	25
Inner Join	26
Outer Joins.....	27
Left Outer Join	27
Right Outer Join	28
Full Outer Join	28
Self Join	29
Cross Join	30
Derived Table	30
Sub Query.....	31
Correlated Sub Query	33
Sql Pivot Sorguları	33
INDEX MİMARİSİ	34
Clustered Index.....	34
Non-Clustered Index	35
Index Fragmentation (Parçalanma)	37
Index Defragmentation (Birleştirme).....	37
Clustered Column Store Index	38
Non-Clustered Column Store Index.....	39
VIEWS.....	40
USER DEFINED FUNCTIONS	42
SCALAR-VALUED FUNCTİONS.....	42
TABLE-VALUED FUNCTİONS.....	44
Stored Procedure.....	44
TRIGGER.....	48
After Trigger	48
Instead Of Trigger	50
TRANSACTION.....	50
SQL SERVER 2016 YENİLİKLERİ	51
DYNAMIC DATA MASKİNG.....	51
NATIVE JSON	53
ROW LEVEL SECURITY	54
TEMPORAL TABLES.....	57

ALWAYS ENCRYPTED	58
LIVE QUERY STATISTICS.....	60

VERİTABANINA GİRİŞ

VERİ (DATA) NEDİR

Çeşitli yöntemlerle depolanabilen, analiz ve raporlama aracı olarak tercih edilen, karakter, sayı, ses, görüntü topluluğudur. Veri günlük yaşamın vazgeçilmez bir parçasıdır. Veri, anlamlandırılarak hayatı yönlendiren için en önemli araçtır. Kişiler veya kurumlar ellerindeki veriler ile geleceklerini devam ettirebilirler. Verinin hem soyut hem somut varlık olduğunu unutmamamız gerekir.

VERİ TABANI (DATABASE) NEDİR

Veriyi dijital ortamda en basit şekilde bir metin belgesiyle bile yönetebiliriz. Ancak Verinin karakter, sayısal, ses, görüntü gibi varlıklar olduğundan bahsetmiştik. Veriyi ilgili formatlarda düzenli bir şekilde saklayabilmemiz için bilgi kümelerine ihtiyacımız vardır. İşte veritabanı, veritabanı yazılımlarıyla veriyi düzenli bir şekilde kümeler (Table) içerisinde, ayrıştırarak (column) ilgili formatlarda depoladığımız ortamdır. Buna Relational Database Management System (RDBMS)'de denir. İlişkisel veritabanlarında veriler tablolar ve sütunlar içerisinde saklanarak kullanıma açılırlar

Genel olarak bilgisayar ortamında metin belgesi, office araçları, vb programları basit veritabanları olarak kabul edebiliriz. Ancak daha güvenli, paylaşılabilen çoklu kullanıcılar için olanak sağlayabilen ortamlara veritabanı demek daha doğru olacaktır.

VERİTABANI SUNUCULARI (DATABASE SERVER)

Veritabanlarını yönetmek, barındırmak için kullanılan yazılımlardır. Bu yazılımlar sayesinde bütün bilgisayar kullanıcıları değil, sadece veritabanı kullanıcıları tarafından veriye erişilerek güvenliği en üst seviye de sağlanır.

MS-SQL SERVER (MICROSOFT SQL SERVER)

Microsoft Sql Server dünyada yaygın olarak kullanılan, Microsoft tarafından geliştirilen bir veritabanı yönetim sistemi yazılımıdır. MS-SQL Server ile veritabanları oluşturabilir, veritabanı kullanıcıları oluşturabilir ve yetkilendirebilirsiniz. Ayrıca veritabanlığının sağlayabilecek diagram oluşturarak da verileri tutarlı bir şekilde depolayabilirsiniz. Verilerinizi yedekleyerek veri güvenliğini sağlayabilirsiniz. MS-SQL Server ile veritabanı yönetimi yapıldığı gibi veriler üzerinde analiz ve raporlamalarda yapabilirsiniz.



MS-SQL SERVER SÜRÜMLERİ

MS-Sql Server kullanırken ihtiyacınıza göre aşağıdaki versiyonlardan bir tanesini seçebilirsiniz.

Sürüm	Açıklama
Express	Max. 10 GB disk boyutuna kadar veritabanı desteği veren ücretsiz sürümüdür.
Developer	Sadece geliştirmek ve tüm özelliklerini öğrenmek için kullanılan ideal bir sürümdür. Hiç bir şekilde ticari olarak kullanılamaz. Bu sürümü de Express sürümü gibi ücretsizdir.
Standart	Enterprise sürümünün alt sürümüdür. Instance başına 128 GB bellek, İş zekâsı, İleri düzey güvenlik, Raporlama gibi özellikler sunar.
Enterprise	En üst sürümüdür. Kurumsal işletmeler için büyük veri merkezi, İş zekâsı, İleri düzey güvenlik, Instance başına disk boyutu kadar hafıza, Maximum 524 PetaByte veri

yönetilebilir.

NORMALİZASYON

Normalizasyon bir ilişkisel veritabanı yaklaşımıdır. Her bir verinin satır ve sütunlarda, ayrıca verilerin ayrıştırılmış tablolarda saklanması destekleyen bir yaklaşımdır. Örnek olarak bir e-ticaret yazılımının veritabanındaki müşteri ve adres bilgilerinin saklandığını düşünelim. Bir müşterinin birden fazla adresi olacağını varsayacak olursak (Fatura ve Teslimat için) Musteriler{ MusteriNo,KullaniciAdi,Email }, Adresler {AdresNo,AcikAdres,İl,İlçe, Musteri} gibi tanımlamalıyız. Bu şekilde verileri ayrıştırarak data anormalliğini gidermiş oluruz. Böyle bir müşterinin 2 farklı adresi için adres tanımlarken tekrar müşteri bilgisi eklememiş oluruz. Normal Form kuralına uymuş olarak veri bütünlüğü sağlamış oluruz.

Normalizasyon kavramı Edgar Frank Codd tarafından geliştirilmiştir. Bu yaklaşım sayesinde sütun bağımlılıkları ve veri bütünlüğü sağlanmış olur. Normalizasyon yaklaşımları 1NF, 2NF, 3NF 'dir.

1NF

Yinelenen satır ve sütun yapısı olmamalıdır. Her tabloda anahtar alan denilen birincil anahtar alanlar tanımlanmalıdır ve bütün alanlar sadece o alana bağımlı olmalıdır. Burada anahtar alanın tanımlanma amacı veriyi diğer verilerden ayırmak içindir. Anahtar alanları Tc. Kimlik numaramız gibi düşünebilirsiniz. Ayrıca anahtar alanlara **Primary Key** denilir.

2NF

Bir tablonun 2NF olabilmesi için 1NF'e uygun olması gerekir. Ayrıca tabloda eğer 2 adet anahtar alan varsa bu kısmi bağımlılık durumu olur. 2.NF'de kısmi bağımlılık ortadan kaldırılmalıdır. Buna göre normalizasyon tanımında yaptığımız gibi Musteriler ve Adresler tabloları oluşturularak tablolar arasında ilişki oluşturulur. 2NF uygulanmasının amacı veriye tek bir alan üzerinden erişimdir. İki tablo üzerinde ilişki genelde ikincil anahtar ile sağlanır(**Birincil anahtarlarda olabilir**). Bu ikincil anahtarlara **Foreign Key** denir.

3NF

Tablonun 3NF olabilmesi için 2NF'e uygun olmalıdır. 3NF'de anahtar olmayan bütün alanlar iki birincil alan ile bulunuyorsa buna geçişken bağımlılık denir. Yani Özellik, Ürün ve ÜrünÖzellik tablolarımızı düşünelim. Ürün => ÜrünÖzellik <= Özellik şeklinde bir bağlantı sağlanır. Bu yaklaşımda da geçişken bağımlılık ortadan kaldırılmalıdır.

Bu 3 yaklaşımın temel amacı veri tekrarının önüne geçmek için kullanılır. Yani veri tekrarını engelleyene kadar tablolarımızı ayırmamızı ister.

SQL SERVER MANAGEMENT STUDIO

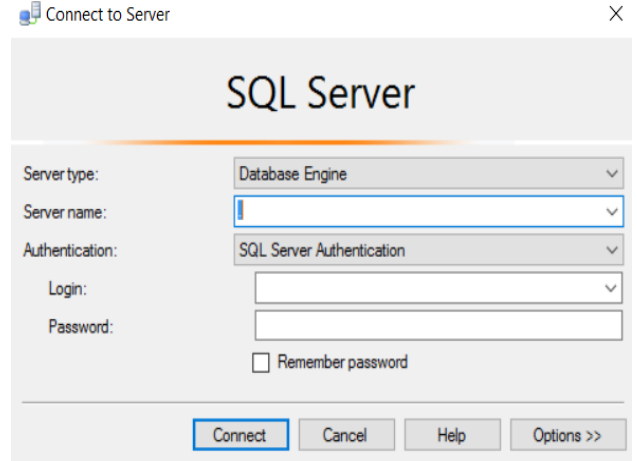
Sql Server Management Studio'yu Sql server'a aktif olarak kullanabilmek için kullanılan IDE olarak tanımlayabiliriz. Sql Server ile Management Studio çok farklı şeylerdir. Sql Server bilgisayarımızda hizmet veren bir sunucu(Server) yazılımı, Management Studio ise bu sunucu yazılımını kullanabilmemiz için kullanılan bir istemci (Client) yazılımdır. Yani bilgisayarımızda sql server olmasa bile, management studio ile başka bir sql server sunucusuna bağlanabiliriz. Management Studio ile Sql Server'a bağlanmak için Başlar => Programlar => Sql Server => Management Studio adımlarını izleriz. İlk olarak karşımıza gelen pencere Connect to Server penceresidir. Connect to Server Penceresi Management studio üzerinden Sql Sunucumuza bağlanmak için kullandığımız bağlantı penceresidir

SERVER TYPE

Sql sunucusunun hizmet tipidir. Mevcut hizmetler; Raporlama Servisi, Analiz Servisi, Entegrasyon servisi ve Database Engine'dir. Veritabanı işlemlerimiz için Database Engine seçilir

SERVER NAME

Bağlantı yapacağımız sunucu adresimiz. Sql Server kurulumu yaparken belirlediğimiz bir isimdir. Ayrıca daha bahsettiğimiz bilgisayarımızda olmayan bir server adıda olabilir



AUTHENTICATION

Bağlantı türümüzdür;

Sql Server Authentication

Daha önce tanımlanmış olan sql kullanıcıları için kullanılan bir seçenektir. Bu bağlantı türünü seçtiğimiz de mevcut kullanıcılar user name ve password bilgilerini girerek sql server'a bağlanabilirler. Eğer sql server kurulumunda mixed mode seçilmiş ise, tanımlı olan **sa** (system admin) kullanıcısına verilen şifre ile bağlanılır. Daha sonra sql serverda kullanıcı oluşturarak bu seçenek kullanılabilir. Yetki bazlı kullanımda ve başka bir servera bağlanmak için çok önemlidir

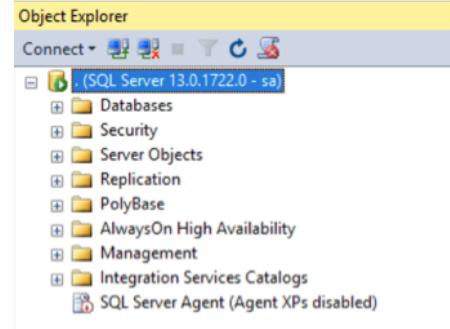
Windows Authentication

Sadece mevcut windows kullanımız ile login olduğumuz bağlantı türüdür.

OBJECT EXPLORER

Sunucuya bağlandığımızda karşımıza ilk çıkan Object Explorer penceresiyle Database Engine hizmetlerinden faydalanabiliriz.

NOT : Bu bölümde geliştirici olarak Databases sekmesinde çalışırız



NEW DATABASE

Databases sekmesine sağ tık yaparak yeni bir veritabanı oluşturabiliriz. Karşımıza gelen new database penceresi seçenekleri

DATABASE NAME : Veritabanı ismi

OWNER : Veritabanı kullanıcısı

LOGICAL NAME : Database dosya isimleridir. Değiştirilebilir ama veritabanı ile aynı isimde olması tavsiye edilir

FILE TYPE : Dosya tipimizi belirler. SQL'de dosya tipi ikiye ayrılır.

1. MDF (Meta Data File)
2. LDF (Log Data File)

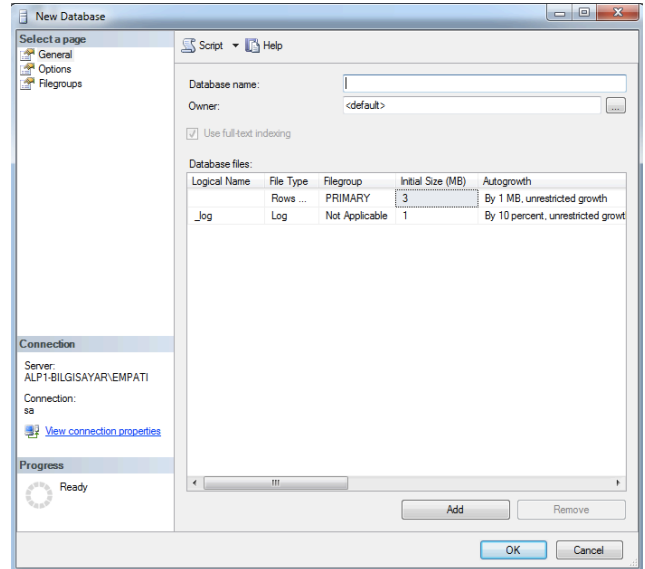
Mdf : Veritabanına girmiş olduğum verilerin aslı tutulur. Yani veritabanına eklediğim bilgilerin kendisi tutulur.

Ldf : Log dosyasıdır. Veritabanında yapılan işlemlerin loglarını tutar. (Şu kullanıcı şu ip'den giriş yaptı, şu bilgiyi güncelledi gibi..)

INITIAL SIZE : Açılacak olan dosyanın başlangıçta kapasitesinin ne kadar olacağını sorar. Burda ki ilk değerler varsayılan değerdir. İstersek değiştirebilir veya maximum bir limit belirleyebiliriz

AUTOGROWTH : Otomatik artış değeridir. Veriler veritabanı boyutunun sınırına ulaştığında bu alanda belirlenen değer kadar veritabanı boyutu arttırılır.

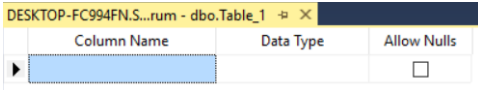
Database ismine SqlOgreniyorum diyelim ve diğer seçeneklere dokunmadan Ok düğmesine tıklayarak veritabanımızı oluşturalım.



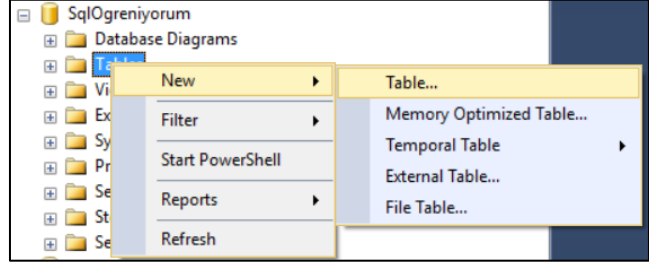
Dikkat : Bu kitap size yardımcı bir kaynak olarak sunulmuştur. Buradaki bütün örneklerimizi bu oluşturduğumuz "SqlOgreniyorum" veritabanı üzerinden sql diliyle yapacağız. Örneklerimizin mantığını anlamak için adımları birebir (sırasıyla) uygulamaya özen gösterelim. Ayrıca yapacağımız her sorgu SQLOgreniyorum veritabanı üzerinden yapılacaktır.

NEW TABLE

Oluşturduğumuz veritabanının solundaki + iconuna tıkladığımızda veritabanımız altındaki sekmeleri göreceğiz. Burada Tables sekmesinde sağ click yaparak New > Table seçeneğine tıklayalım ve aşağıdaki pencereyi inceleyelim



Column Name	Data Type	Allow Nulls
		<input checked="" type="checkbox"/>



Buradaki **Column Name** sütun adıdır. Tabloyu oluşturan en önemli unsurlar

sütunlar (kolonlar) olduğu için sütun ismini veriyoruz. Daha sonra **Date Type** alanını incelediğimizde bu sütuna girilecek veri tipini belirliyoruz. Son seçeneğimiz ise **Allow Nulls** bu alan boş geçilebilir mi bunu belirliyoruz. Eğer sütuna girilecek veri önemli ise kesinlikle seçilmemelidir. Bu seçimi yapmak bu alanın boş geçilebilir olduğunu gösterir. İlk bölümde tablo oluşturmaya kod ile devam edeceğiz. Kitapta ağırlıklı olarak sql üzerinde duracağız.

CONSTRAINTS

Constraint tablolaradaki alanlara girilen verilerin kontrollerini yapan ve kısıtlamalar getiren tekniklerin bütünüdür. Constraint ile amaçlanan tamamen veri bütünlüğünü, doğrulamasını ve tutarlılığını sağlamaktır.

Primary key, Foreign Key, Unique Key, Check ve Default Constraintlerdir. T-SQL bölümünde Constraint örnekleri anlatılacak ve örneklendirilecektir.

SQL SERVER VERİ TİPLERİ

Veri tipleri tablolarda saklanacak verilerin hangi formatta depolanacağını belirlediğimiz t-sql tipleridir. Daha önce verinin bir ses veya görüntü,metin, sayı gibi değerler olduğunu söylemiştik. İşte bir veritabanının dosya sisteminden ayıran en büyük özelliklerden biriside veri tipleridir. Sql server bize veri tipleri konusunda güçlü bir destek veriyor. Veri tiplerini metinsel, sayısal, ondalıklı, mantıksal, tarihsel tipler olarak inceliyoruz. Aşağıda sık kullanılan bu veri tipleri tablo halinde listelenmiştir

Tam sayı veri tipleri

Tam Sayılar	Ondalıklı Sayılar	açıklama
bit	1 ile 0	Bool olarak kullanılır. True ve False
tinyint	0 ile 255	
smallint	-2 ¹⁵ (-32,768) ile 2 ¹⁵ -1 (32,767)	
int	-2 ⁶³ (-9,223,372,036,854,775,808) ile 2 ⁶³ -1 (9,223,372,036,854,775,807)	
bigint	-2147483648 ile 21 47483647 arası tam sayı	

Ondalıklı sayı veri tipleri

Tür	Değer aralığı	açıklama
real	- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 ile 3.40E + 38	-
float	- 1.79E+308 ile -2.23E-308, 0 and 2.23E-308 ile 1.79E+308	-

Tarih ve Saat veri tipleri

Tür	açıklama
Datetime	Tarih Saat-Saniye
Datetime2	Tarih saat saniye sadise
Smalldatetime	Tarih ve Saat
Date	Tarih
Time	Saat ve Saniye

Metinsel veri tipleri

Tür	Değer aralığı	Açıklama
Char	Max. 8000 karakter	Char(n) girilen karakter kadar değil n sayısı kadar alanı kapsar. Unicode karakterleri desteklemez
Varchar	Max. 8000 karakter	Varchar(n) girilen karakter kadar sayısı kadar alanı kapsar. Uniocede karakterleri desteklemez
Nchar	Max. 4000 karakter	NChar(n) girilen karakter kadar değil n sayısı kadar alanı kapsar. Unicode karakteri destekler
Nvarchar	Max 4000 karakter	Nvarchar(n) girilen karakter sayısı kadar alanı kapsar. Unicode karakterleri destekler

TRANSACT SQL (T-SQL)

SQL (Structured Query Language)

SQL (Structured Query Language) 1970'li yıllarda geliştirilen veritabanı yönetim sistemleri dili olan **SEQUEL** dilinin ardılıdır. SEQUEL dili 1980 li yılların sonunda SQL olarak adlandırıldı ve **ANSI** tarafından standartlaştırıldı. SQL dili, veritabanı yönetim sistemleriyle iletişime geçen, verileri temel anlamda yönetmek için kullanılan bir sorgu dilidir. SQL ile tablolarımıza veri ekleme, listeleme, güncelleme, silme işlemleri yaparız.

T-SQL

T-sql sql dilinin sql servera uyarlanmış halidir. Sql dili temel sorgu standartlarını belirlediği bir yapıdır. T-sql ise Sql'i referans alan ve daha gelişmiş bir dildir. T-sql Microsoft Sql Server ile kullanılan bir sorgu dilidir. T-sql dilini genellikle 3 başlık altında inceleriz. Bunlar;

1. Data Definition Language (DDL) = (create, alter, drop)
2. Data Manipulation Language (DML) = (select, insert, update, delete)
3. Data Control Language (DCL) = (grant, deny, revoke)

olarak incelenir. Hepsi t-sql dilidir fakat çalışma mantığına göre sınıflandırılmışlardır.

T-SQL İLE KISITLAYICILARI KULLANARAK VERİ BÜTÜNLÜĞÜNÜ SAĞLAMA

Bu bölümde t-sql kullanarak daha önce oluşturduğumuz SqlOgreniyorum veritabanına tablo oluşturacağız ve constraintleri kullanarak veri bütünlüğünü ve tutarlılığını sağlayacağız.

Örnek : Şimdi New Database seçeneği ile yeni bir veritabanı oluşturalım ve name kısmına “SqlOgreniyorum” diyelim. Ardından Management Studio ortamında toolbar’dan **New Query Düğmesiyle** yeni bir sorgu penceresi açalım



TABLO OLUŞTURMA

Önceki açılan query penceresine aşağıdaki komutları yazıyor sonrasa seçip çalıştırıyoruz

Calisanlar ve Departman tablolarını oluşturuyoruz. Daha sonra bu iki tabloyu ilişkilendireceğiz

go

create table Calisanlar

(

CalisanID int not null,

Adi nvarchar(50) not null,

SoyAdi nvarchar(50) not null,

Email nvarchar(11) not null,

TcNo nvarchar(11) not null,

Maas money not null,

DepartmanID int not null

)

go

create table Departman

(

DepartmanID int not null,

DepartmanAdi nvarchar(50) not null

)

İlk olarak Calisanlar tablosundaki bütün alanların fonksiyonel bağımlı olacağı bir primary key tanımlamamız gerekiyor. Bunun nedeni ilişkisel veritabanında ilk kural satırların unique olması gerekir.

Adi,SoyAdi,Email,TcNo,Maas ve DepartmanID alanları üzerinde değerlendirme yapacak olursak, ilk olarak Adi ve SoyAdi ve Maas alanlarını elememiz gerekiyor. Bu alanlarda ortak değerler olabilir. Yani ikitane Kamil ve 3000 TL maaş olan farklı personeller olabilir. TcNo,Email, ve CalisanID alanları pk için uygun alanlardır. Fakat değerler sıralı olsun diye CalisanID üzerinden gidiyoruz.

PRIMARY KEY CONSTRAINT

Primary Key kısıtlayıcısı alanın boş geçilmesini engeller ve değerinin benzersiz olmasını sağlar.Tıpkı Tc Kimlik Nolarımız gibi. Primary Key’de verinin TC Nosu’dur

Primary key tablo ilk oluşturulduğu zaman veya daha sonra alter table diyerek tabloya eklenebilir. Ancak daha sonra primary key oluştururken tabloda veri olmaması gerekir.

Primary Key ve Constraint Key

alter table Calisanlar

add constraint pkCalisanKey

```
primary key(CalisanID)
```

Primary key tanımlayarak artık CalisanID alanın benzersiz olmasını sağladık ve her bir sütunun bu alana bağlı olmasını sağladık.

UNIQUE CONSTRAINTS

Bir tabloda primary key kullanarak sadece bir tane benzersiz alan elde edebilirsiniz. Primary key tanımlamaktaki amaç alanı benzersiz yapmak değil bütün alanların o alana bağımlı olmasını sağlamaktır. Unique Key ise alanı sadece benzersiz yapmak için kullanılır.

Dikkat : Unique Key Primary Key'in aksine birden fazla olabilir.

Aşağıdaki kod ile Email ve TcNo alanlarını unique key olarak tanımlıyoruz.

Unique Key

```
alter table Calisanlar
add constraint uniqueEmail unique (Email),
constraint uniqueTcNo unique (TcNo)
```

FOREIGN KEY CONSTRAINT

İkincil anahtarlar ilişkisel tablolar içerisindeki alanlar için veri tutarlılığını ve bağımlılığını sağlamak için kullanılır. Foreign key olarak tanımlanan alan için kesinlikle referans alacağı bir alan ve tablo belirtilmek zorundadır.

Aşağıdaki komut ile daha önce oluşturduğumuz çalışan ve departman tablolarıyla DepartmanID 'ler üzerinden bir ilişki kuruyoruz. Öncelikle gidip Departman tablosundaki departmanID alanını primary key olarak tanımlayalım.

Primary Key Tanımlaması

```
alter table Departman
add constraint pkDepartmanKey primary key(DepartmanID)
```

Daha sonra tekrar Calisanlar tablomuzu düzenleyerek DepartmanID alanını foreign key yapıyoruz.

Foreign Key Tanımlaması

```
alter table Calisanlar
add constraint fkDeparmantID
foreign key(DepartmanID) references Departman(DepartmanID)
```

Komutu seçip çalıştırdığınızda artık bu iki alan üzerinden Calisanlar ve Departman tablosu arasında ilişki sağlamış olduk. Artık departman eklenmeden hiç bir şekilde çalışanın DepartmanID'sine veri ekleyemeyiz. Yine aynı şekilde Calisan tablosunda o departmana ait bir veri varken departman silemiyoruz. İş tam burada veri bütünlüğünü sağlayarak veri tutarsızlığının önüne geçmiş oluyoruz.

Dikkat : Foreign key olarak tanımlanan sütun veri tipi ile referans alınan sütun tipi birbirine eşit olmak zorundadır

CHECK CONSTRAINTS

Check Constraint tablonun ilgili sütununa veri girerken veriyi belirlediğimiz bir stilde kontrol eder. Örneğin çalışanların TcNo'sunu 11 karakterden az olmamasını sağlayalım.

Check Constraint

```
alter table Calisanlar
    add constraint tcNoKontrol
    check(Len(TcNo) = 11)
```

Yukarıdaki komutu çalıştırdığımızda artık Tcno alanının değerini min ve max 11 karakter olacak şekilde kontrol edecektir.

DEFAULT CONSTRAINTS

Default constraint bir sütuna değer girilmediğinde o sütun için default bir değer atanması için tanımlanır. Örneğin çalışanlar tablosuna GirişTarihi sütunu ekleyelim ve değerinde kayıt oluştuğunda girilmesini sağlayalım.

Default Constraints

```
alter table Calisanlar
    add IseGirisTarihi date,
    constraint defaultDeger default(getdate()) for IseGirisTarihi
```

Bu bölümde constraint'leri kullanarak tablodaki sütunlara girilecek veriler için kısıtlamalar getirerek veri tutarlılığını sağlamaya çalıştık. Sizlerde t-sql kullanarak farklı tablolar oluşturup constraintleri çalıştırınız.

TABLOYA VERİ EKLEME (INSERT)

Bu bölümde t-sq ile query penceresini kullanarak daha önce oluşturduğumuz SqlOgreniyorum veritabanındaki tablolarımıza kayıt ekliyoruz. Bir sonraki bölümde ise select ifadesi ile tablo üzerinde sorgulama yapacağız

İlk olarak departman tablosuna kayıt eklemek için aşağıdaki komutları seçerek execute ediyoruz

Insert İşlemi

```
insert Departman
(DepartmanID,DepartmanAdi)
values
(1,'Yazılım')
```

```
insert Departman
(DepartmanID,DepartmanAdi)
values
(2,'Grafik')
```

```
insert Departman
(DepartmanID,DepartmanAdi)
```

values

(3,'Muhasebe')

Daha sonra Çalışanlar tablosuna kayıt eklemek için komutlarımızı seçerek execute ediyoruz

Insert İşlemi

insert Calisanlar

(CalisanID, Adi, SoyAdi, Email, TcNo, Maas, DepartmanID)

values

(1,'Ekrem','Yıldırım','e@ba.com','12345567898',1000,1)

insert Calisanlar

(CalisanID, Adi, SoyAdi, Email, TcNo, Maas, DepartmanID)

values

(2,'Alp','Kurtboğan','a@ba.com','23345567898',3500,1)

insert Calisanlar

(CalisanID, Adi, SoyAdi, Email, TcNo, Maas, DepartmanID)

values

(3,'Rıdvan','Aksoy','r@ba.com','33345567898',500,2)

insert Calisanlar

(CalisanID, Adi, SoyAdi, Email, TcNo, Maas, DepartmanID)

values

(4,'Mert','Kurt','m@ba.com','43345567898',1500,2)

Dikkat : insert komutu bir tabloya veri eklemek için kullanılan bir dml komutudur. Öncelikle hangi tabloya ve hangi alanlara kayıt ekleyeceğimizi belirleriz.

T-SQL SORGULAMA ÖRNEKLERİ

Bu bölümde Calisanlar ve Departman tablolarına yönelik select ifadeleri üzerinde duracağız. Daha sonra operatörler, filtreleme, sıralama,gruplama, takma isimler konuları ile devam edeceğiz.

SELECT KOMUTU

Select ifadesi bir tablodaki kayıtlar son kullanıcıya sunmak için kullanılan t-sql komutudur. Aşağıdaki sorguyu sağdan okuyarak inceleyelim. Sorgudaki from Calisanlar komutu (Calisanlardan) * komutu (Bütün Sütunları) select komutu ise (seç) olarak yorumlanır. Sorguyu seçip execute ettiğimizde karşımıza query penceresinin hemen altında Results penceresi gelir ve tablodaki verileri bize sanal (gerçek olmayan) bir tablo olarak sunar. Bu tablo üzerinde değişiklik yapabileceğimiz bir tablo değildir. Sadece verileri okuyabildiğimiz **read only** bir tablodur.

Select Sorgusu

select *

```
from Calisanlar
```

Eğer tablodaki bütün sütunları değil de sadece belirtilen sütunları istiyorsak o zaman sorgumuz aşağıdaki gibi olmalıdır.

Select Sorgusunda Kolon Filtreleme

```
select
    Adi,
    SoyAdi,
    Email,
    Maas
from Calisanlar
```

Yukarıdaki sorgu sonucunda Results tablosunda Adi,SoyAdi,Email,Maas alanları sonuç olarak dönecektir.

ALİASES (TAKMA AD)

Aliases result set'te bir tablonun veya sütunun ismini anlık olarak değiştirmek için kullanılır. Özellikle kolon isimlerinin daha anlaşılır olması , tablo isimlerinin kısaltılması için tercih edilir. Aşağıdaki örneği inceleyelim

Aliases Kullanımı

```
select
    Adi,
    SoyAdi as [Soy Adı]
from Calisanlar
```

Yukarıdaki sorguyu çalıştırdığımızda Result penceresinde SoyAdi sütunu Soy Adı olarak geldi.

Dikkat : T-sql de kolon isimler ve tablo isimler arasında boşluk bırakılmaz. Köşeli parantezler [] boşluk bırakırsak veya bırakıldıysa o boşlukları devre dışı bırakmak için kullanılır

Şimdi tekrar Calisanlar tablosundaki Adi ve SoyAdi alanlarını getirelim fakat iki kolonu birleştirerek sonuç dönmesini sağlayalım

Concat Kullanımı

```
select
    Concat(Adi,' ',SoyAdi)
from Calisanlar
```

Yukarıdaki örnekte Concat komutu iki ifadeyi birleştirmek için kullanılır. Ama unutmamız gereken şey bu değişiklik sadece result penceresinde olur. Bu sorguyu çalıştırdığımızda tek bir kolon gelecek ve ismi No Column Name olacaktır. İşte bu durumlarda son kullanıcı bunun ne olduğunu anlamaya bilir. Sorguyu aşağıdaki gibi çalıştırırsak artık anlık yeni bir isim vermiş oluruz.

Concat Kullanımı

```
select
    Concat(Adi,' ',SoyAdi) as [Adı ve SoyAdı]
```

from Calisanlar

WHERE KOMUTU

Where komutu tablodaki satırları filtrelemek için kullanılır. Result penceresi sadece sonuç kümesidir. Burada where komutu ile kriter'e uyan veriler true olarak kabul edilir ve sadece o satırlar sonuç olarak döndürülür.

Çalışanlar tablosundan sadece CalisanID'si 1 olan satırı getirelim

Where Komutu

```
select * from Calisanlar
where CalisanID = 1
```

Yukarıdaki sorguyu çalıştırdığımızda Calisanlar tablosundan CalisanID'si 1 olan çalışanın bütün alanlarına erişmiş oluruz. Tekrar aşağıdaki sorguda CalisanID'si 10 olan satıra erişelim.

Where Komutu

```
select * from Calisanlar
where CalisanID = 10
```

Yuradaki sorguyu çalıştırdığımızda result set penceresinin boş olduğunu göreceğiz. Çünkü Calisanlar tablomuzda böyle bir kayıt yok.

Dikkat : Where komutu ile sadece kriter'e uyan kayıtlar listelenir

WHERE KOMUTU OPERATÖRLERİ

Bu operatörler where komutu ile veri filtrelemek için kullanılır. Açıklamalarına bakalım ve daha sonra örnekler üzerinden gidelim.

Karşılaştırma Operatörleri

- Eşittir (=)
- Büyüktür (>)
- Küçüktür (<)
- Büyük Eşit (>=)
- Küçük Eşit (<=)
- Eşit değil (!= veya <>)

Mantıksal Operatörler

- And
- Or

Diğer Operatörler

- Between
- In
- Like

Eşitlik operatörünü daha önceki where bölümünde görmüştük. Şimdi diğer operatörlerimizi inceleyelim.

Maaşı 500'den büyük olan çalışanlarımız

Where Sorgusu Operatör Örnekleri

```
select * from Calisanlar
```

```
where Maas > 500
```

Maası 500'e eşit veya 500'den büyük olan çalışanlarımız

Where Sorgusu Operatör Örnekleri

```
select * from Calisanlar
```

```
where Maas = 500 or Maas > 500
```

Buradaki sorguyu mantıksal operatörle birleştirdiğimiz gibi karşılaştırma operatörlerinden büyük eşit operatörü ile de kullanabilirdik

Where Sorgusu Operatör Örnekleri

```
select * from Calisanlar
```

```
where Maas >= 500
```

Maası 500'den küçük veya eşit olan çalışanlarımız

Where Sorgusu Operatör Örnekleri

```
select * from Calisanlar
```

```
where Maas <= 500
```

Where komut ile sadece bir alan üzerinde filtreleme yapabildiğimiz gibi istersek mantıksal operatörler ile farklı alanlar üzerinde de filtreleme yapabiliriz.

Where Sorgusu Operatör Örnekleri

```
select * from Calisanlar
```

```
where DepartmanID = 1 and Maas > 2000
```

Yukarıdaki örneğimiz de DepartmanID si 1 ve Maas'ı 2000'den büyük olan çalışan(larımız) gelecektir.

In Operatörü

In operatörü where komutu ile kullanılır. Birden fazla eşitlik durumu yazmak yerine tercih edilir. Öğreniğimizi iki farklı şekilde inceleyelim

In Kullanımı

```
select * from Calisanlar
```

```
where Email = 'e@ba.com' or Email = 'r@ba.com'
```

Yukarıdaki örneğimiz de Calisanlar tablosundan Email alanı e@ba.com veya r@ba.com olanları listeliyoruz. Aynı sorguyu in ile yazalım

In Kullanımı

```
select * from Calisanlar
```

```
where Email in ('e@ba.com','r@ba.com')
```

Yukarıdaki sorgu ile bir öncekini karşılaştırdığımızda in ile sorgunun daha kısa yazıldığını görüyoruz.

Not In Operatörü

In operatörünün tam tersidir. İçinde eşit olan değil de, içinde eşit olmayan anlamına gelir.

Not In Kullanımı

```
select * from Calisanlar
where Email not in ('e@ba.com','r@ba.com')
```

LIKE OPERATÖRÜ

Like operatörü tabloda sütun üzerinden arama yaparak filtrelemek için kullanılır.

Like Kullanımı

```
select * from Calisanlar
where Adi like '%a%'
```

Yukarıdaki sorgumuz adi içinde a geçenleri;

Like Kullanımı

```
select * from Calisanlar
where Adi like 'a%'
```

Yukarıdaki sorgumuz adi a ile başlayanları;

Like Kullanımı

```
select * from Calisanlar
where Adi like '%a'
```

Yukarıda ki sorgumuz adi a ile bitenleri listeleyecektir.

Order By (Sıralama)

Order by komutu sorgu sonucu sıralamak için kullanılır. Order by komutu tablodaki belirtilen alan veya alanlara göre sonucumuzu bize sıralı bir şekilde gösterir. Order by komutu asc (artan) ve desc(azalan) şekilde sıralama olanağı sağlar. Aşağıdaki sorguda çalışanlarımızı maaşlarına göre sıralayalım.

Order By Kullanımı

```
select * from Calisanlar
order by Maas asc
```

Select ile Calisanlar tablosundaki çalışanlar listelenerek maaş alanına göre artan bir şekilde sıralanır

Order By Kullanımı

```
select * from Calisanlar
order by Maas desc
```

Yukarıda ki örneğimiz de çalışanlar tablosundan Maaş'a göre azalan bir sıralama yaptık.

Dikkat : Order by ile asc veya desc sıralama türünü belirmezsek, sıralama default olarak asc olacaktır

Order by komutu ile aliases alanlar da sıralama yapılabilir. Örneğimizde yine Adi ve SoyAdi alanlarını birleştirerek takma isim veriyoruz ve o alana göre sıralama yapıyoruz.

Order By ile Contact Fonksiyonu Kullanımı

```
select  
    Concat(Adi, ' ', SoyAdi) as [Adı ve SoyAdı]  
from Calisanlar  
order by [Adı ve SoyAdı] desc
```

Dikkat : Aliases alanlar sadece order by ile kullanılabilir. Where komutu ile filtrelemeye dahil edilemez. Nedeni bu alan sadece result aşamasında oluşur. Server'a alan üzerinden istek gönderdiğimizde tabloda öyle bir alan olmadığı için sorgumuz hatalı olur. Order by'da ise yine istek gönderilip sıralama cevap aşamasında oluşacağı için as komutu kullanımı hataya neden olmaz.

Distinct (farklı kayıtları elde etme)

Distinct komutu tablodaki aynı kayıtların sonuç kümesinde sadece tekil olarak listeler. Yani distinct komutu tekrar eden kayıtları elimine ederek benzersiz gelmesini sağlar.

Distinct Kullanımı

```
select Distinct DepartmanID from Calisanlar
```

Sonuç kümesini incelediğimiz de DepartmanID alanına göre bir distinct işlemi yaptık. Calisanlar tablomuzda 4 kayıt varken sadece 2 kayıt geldiğini görüyoruz.

Top

Top komutu sorgu sonucunda satır seçmek için kullanılır. Örnek olarak çalışanlar tablomuzdan 2 kayıt getirelim.

Top Kullanımı

```
select top 2 * from Calisanlar
```

Sorgu sonucunu CalisanID üzerinden inceleyelim. Eklenme sırasına göre kayıtların seçildiğini görüyoruz. Şimdi çalışanlar tablosunu maaş alanına göre sıralayım ve en az maaş alan 2 çalışanımızı getirelim

Top ile Order By Kullanımı

```
select top 2 * from Calisanlar  
order by Maas asc
```

Yukarıdaki örneğimizin sonucunu incelediğimizde tekrar CalisanID alanına dikkat edecek olursak 3 ve 1 olarak sıralandı. Bu sefer satır seçerken eklenme sırasına göre değil maaş alanına göre işlem yaptık. Yani önce maaşa göre sıralama daha sonra da top ile satır seçtik.

Top kullanımında WITH TIES komutu oldukça önemlidir. With Ties komutunu incelemeden Calisanlar tablomuzda yeni bir çalışan ekleyelim;

Insert Kullanımı

```
insert Calisanlar  
(CalisanID, Adi, SoyAdi, Email, TcNo, Maas, DepartmanID)
```

values

(5, 'Sabri', 'Sariolu', 's@gs.com', '98765432191', 500, 3)

Şimdi maaş alanına göre artan bir sıralama yapalım

Order By Kullanımı

select * from Calisanlar

order by Maas asc

Sonucu incelediğimizde maaşı 500 olan Rıdvan ve Sabri'yi görüyoruz. Şimdi tekrar sorgumuzu top 1 ile tekrar yazalım ve en düşük maaş alan çalışanımızı bulalım.

Order By ve Top Kullanımı

select top 1 * from Calisanlar

order by Maas asc

Sonuç olarak incelediğimizde sadece 1 çalışanın geldiğini görüyoruz. Aslında en düşük maaş değeri 500 idi. Fakat listeleme yaparken ilk eklenen veriyi getirdi. İşte burada sorgumuzu with ties ile tekrar yazıyoruz.

With Ties Kullanımı

select top 1 with ties * from Calisanlar

order by Maas asc

Sonucu incelediğimizde top 1 dememize rağmen 2 satırın geldiğini gördük. Burada TIES komutu bağlaç görevi görüyor. Yani sıralanan alanı referans alarak top n kullanımında son satırdaki sıralanan alan ile benzer kaydı getirir. Bu şekilde çalıştırdığımızda Rıdvan ve Sabri değerlerini görüyor olacağız.

AGGREGATE FUNCTIONS

Aggregate functionlar select ifadesiyle kullanılan geriye tek hücre olarak sonuç dönen fonksiyonlardır. Aşağıdaki sorguları inceleyelim.

Aggregate Funtion Örnekleri

select COUNT(*) from Calisanlar -- bütün satırları sayar.

select COUNT(CalisanID) from Calisanlar -- CalisanID alanına göre null olmayan satırları sayar

select SUM(Maas) from Calisanlar -- maaş verilerini toplar

select AVG(Maas) from Calisanlar -- maaş verilerinin ortalamasını verir

select Min(Maas) from Calisanlar -- maaş verilerinden minimum olanını verir

select MAX(Maas) from Calisanlar -- maaş verilerinin maximum olanını verir

CASE WHEN

Case When ifade select içerisinde koşullu ifadeler için kullanılır. Yani koşula göre sonuç dönmesini sağlar. Örneğimizde DepartmanID alanı 1 olanlar için Yazılım, 2 olanlar için Grafik, 3 olanlar için Muhasebe yazmasını istiyoruz

Case When Örneği

```
select
    Adi,
    SoyAdi,
    DepartmanID,
    case DepartmanID
        when 1 then 'Yazılım'
        when 2 then 'Grafik'
        when 3 then 'Muhasebe'
    end as KosulluSütun
from Calisanlar
```

GROUP BY (GRUPLAMA)

Group by komutu tablodaki ortak verilerin gruplanarak dönmesini sağlar. Group by kullanımında önce gruplanacak alan belirlenip daha sonra o alana göre bir select işlemi yapılır. Sütundaki ortak olan değere göre alanlar gruplanır. Özellikle raporlama sorgularında anlamlı sonuçlar elde etmek için çok sık kullanılır. Örneğimizi inceleyelim;

Group By Kullanımı

```
select DepartmanID from Calisanlar
group by DepartmanID
```

Yukarıdaki örnekte Calisanlar tablosunu DepartmanID alanına göre grupladık ve DepartmanID alanını göre select yaptık. Sonucu incelediğimizde distinct ile aynı sonuca ulaşırız.

Dikkat : Group by komutu gruplanan alana göre alt kümeler oluşturarak sonuçların tek bir alan üzerinde görüntülenmesini sağlar.

DepartmanID alanına göre gruplama yaptığımızda sonucu bu şekilde düşünebiliriz

DepartmanID							
1							
2							
3							

CalisanID	Adi	SoyAdi	Email	TcNo	Maas	DepartmanID	IseGirisTarihi
3	Rıdvan	Aksoy	r@ba.com	33345567898	500.00	2	2017-02-26
4	Mert	Kurt	m@ba.com	43345567898	1500.00	2	2017-02-26

CalisanID	Adi	SoyAdi	Email	TcNo	Maas	DepartmanID	IseGirisTarihi
1	Ekrem	Yıldırım	e@ba.com	12345567898	1000.00	1	2017-02-26
2	Alp	Kurtboğan	a@ba.com	23345567898	3500.00	1	2017-02-26

CalisanID	Adi	SoyAdi	Email	TcNo	Maas	DepartmanID	IseGirisTarihi
5	Sabri	Sanolu	s@gs.com	98765432191	500.00	3	2017-02-26

Şimdi aynı sonucu referans alarak aggregate functionlar ile detaylarına inelim.

GROUP BY VE COUNT FONKSİYONU

Count fonksiyonu tablodaki satırları saymak için kullanılır.

Group By ve Aggregate Function Kullanımı

```
select
    DepartmanID,
```

```

COUNT(*) as CalisanSayisi
from Calisanlar
group by DepartmanID

```

Sonucu incelediğimizde her bir departmanID alanının yanında CalisanSayisi alanı görüyoruz. Count fonksiyonu ile Alt kümelerin bütün satırlarını sayarak aslında ortak elemanların sayısını görüyoruz.

Group By ve Sum Fonksiyonu

Group By ve Aggregate Function Kullanımı

```

select
    DepartmanID,
    SUM(Maas) as ToplamMaas
from Calisanlar
group by DepartmanID

```

Yukarıdaki sorguda sonucu incelediğimizde DepartmanID alt kümelerindeki Maas alanlarını toplayarak DepartmanID alanının yanında gösterdi. Böylece Departmanlara göre toplam maaş'ı görmüş olduk.

Group By ile Aggregate Function Kullanımı

```

select
    DepartmanID,
    MAX(Maas) as maximumMaas
from Calisanlar
group by DepartmanID

```

Yukarıdaki sorguda sonucu incelediğimizde yine alt kümelerdeki maximum maaşları bulduk. Son olarak bu fonksiyonları hep birlikte kullanalım ve sonucu inceleyelim.

```

select DepartmanID from Calisanlar
group by DepartmanID

```

DepartmanID	CalisanID	Adi	SoyAdi	Email	TcNo	Maas	DepartmanID	IseGirisTarihi
1	1	Ekrem	Yildirm	e@ba.com	12345567898	1000.00	1	2017-02-26
2	2	Alp	Kurtboğan	a@ba.com	23345567898	3500.00	1	2017-02-26
3	3	Rıdvan	Aksoy	r@ba.com	33345567898	500.00	2	2017-02-26
4	4	Mert	Kurt	m@ba.com	43345567898	1500.00	2	2017-02-26
5	5	Sabri	Sanolu	s@gs.com	98765432191	500.00	3	2017-02-26

Group By ve Aggregate Function Örneği

```

select
    DepartmanID,
    Count(*) as CalisanSayisi,
    SUM(Maas) as ToplamMaas,

```

```
Avg(Maas) as OrtalamaMaas,  
MAX(Maas) as maximumMaas,  
Min(Maas) as minimumMaas
```

```
from Calisanlar
```

```
group by DepartmanID
```

DepartmanID	CalisanSayisi	ToplamMaas	OrtalamaMaas	maximumMaas	minimumMaas
1	2	4500,00	2250,00	3500,00	1000,00
2	2	2000,00	1000,00	1500,00	500,00
3	1	500,00	500,00	500,00	500,00

Group by ve Having

Having komutu group by kullanımında filtreleme yapmak için kullanılır. Group by kullanımından sonra where komutu kullanılmaz. Daha önce group by kullanımında var olan tablo yerine grup tablolarının geldiğinden bahsetmiştim. Örneğimizi inceleyelim.

Group By ve Having Kullanımı

```
select DepartmanID from Calisanlar  
group by DepartmanID  
having DepartmanID <> 1
```

Sonucu incelediğimizde DepartmanID si 1 olmayanlar listelenecektir.

INSERT, UPDATE, DELETE İŞLEMLERİ

Bu bölümde dml komutları ile insert,update ve delete işlemlerini inceleyeceğiz.

Insert Komutu

Insert komutunu kitabımızın ilk bölümlerinde oluşturduğumuz tabloya veri eklemek için kullanmıştık. Ancak insert komutunun çeşitli kullanımları da mevcuttur. Örnekleri sırasıyla inceleyerek uygulayalım.

Insert Values

```
insert Departman  
(DepartmanID,DepartmanAdi)  
values  
(5,'İK')
```

Çoklu Insert Yapmak

```
insert Departman  
(DepartmanID,DepartmanAdi)  
values  
(5,'Finans'),
```

(6, 'Teknik Çizim')

T-Sql ile bu şekilde insert(ler) yapabildiğimiz gibi bir tablodaki kayıtları başka bir tabloya eklemeyide destekliyor. Öncelikle Proje çalışanları isimli bir tablo oluşturalım

Tablo Oluşturma

```
create table ProjeCalisanlari
(
    ID int identity(1,1),
    AdiSoyAdi nvarchar(50),
    DepartmanID int
    constraint pkProjeCalisanlari primary key(ID)
)
```

Dikkat : identity(1,1) komutu o sütunun değerinin sql server tarafından otomatik arttırılacağını işaret eder.

Tablomuzu oluşturduktan sonra aşağıdaki komutumuzu çalıştırıyoruz.

Insert select

```
insert ProjeCalisanlari
(AdiSoyAdi, DepartmanID)
select CONCAT(Adi, ' ', SoyAdi), DepartmanID from Calisanlar
where DepartmanID = 1
```

Örneğimizde DepartmanID si 1 olan çalışanlarımızın Adi ve SoyAdi alanlarını birleştirerek ve DepartmanID alanlarını seçerek ProjeCalisanlari tablosuna ekledik.

Identity Column

Identity column sql server tarafından tabloda otomatik artan satır için kullanılır. Özellikle primary keylerin tarafımızdan oluşturulması veri arttıkça yönetilmesi zor hale gelecektir. Daha önce primary keylerin benzersiz olması gerektiğinden bahsetmiştik. Yukarıda bununla ilgili bir örnek yapmıştık. Şimdi biraz daha detaylı inceleyelim.

İlk olarak Atable isimli bir tablo oluşturuyor ve Col1 alanını identity yapıyoruz

Identity Column

```
create table ATable
(
    Col1 int identity(1,1),
    Col2 nvarchar(5)
)
```

Tablomuzu oluşturduktan sonra da aşağıdaki komutlarımız ile insert işlemimizi yapıyoruz

Insert İşlemleri

```
insert ATable
```

```
(Col2)
```

```
values
```

```
('A')
```

```
insert ATable
```

```
(Col2)
```

```
values
```

```
('B')
```

Dikkat ederseniz sadece Col2 sütununa değer ekledik. Col1 değerleri sql server tarafından üretildi. Identity(1,1) ilk ifade 1'den başla ikinci ifade ise bir bir art anlamına gelir. İsterseniz Farklı bir değerden başlatıp, farklı bir şekilde de artırımını gerçekleştirebilirsiniz.

Şimdi identity column olan bir sütun'a değer eklemeye çalışalım.

Identity Column'a Değer Ekleme

```
insert ATable
```

```
(Col1,Col2)
```

```
values
```

```
(4,'C')
```

Msg 544, Level 16, State 1, Line 283

Cannot insert explicit value for identity column in table 'ATable' when IDENTITY_INSERT is set to OFF.

Yukarıdaki örnekte Col1 sütunu identity olduğu için sql server tarafından bize o alan için insert olanağı verilmiyor. Eğer o sütuna veri eklemek istiyorsak **identity_insert** komutunu kullanmalıyız. Identity_insert komutu identity olan alanlar için değer üretmemizi sağlar.

Identity Insert Kullanımı

```
set identity_insert ATable on
```

```
insert ATable
```

```
(Col1,Col2)
```

```
values
```

```
(101,'E')
```

```
set identity_insert ATable off
```

Delete Komutu

Delete bir tablodan satır veya satırları silmek için kullanılır. Özellikle tüm satırı etkileyeceği için dikkatli kullanılması gereken bir komuttur. Örneğimiz daha oluşturduğumuz Atable verilerini silelim

Tablodan Verileri Silmek

```
delete from ATable
```

Komutu çalıştırdığımızda tablodaki verilerimiz tamamen silindi. Ancak tekrar bir insert işlemi yaparsak Col1 alanın değeri identity olduğu için yeni eklenecek değer en son üretilen değerden bir fazlası olacaktır. Buradan anladığımız verilerimiz tablodan kalıcı olarak siliniyor ama transaction-logda kayıtlı olarak kalıyor. Yani delete komutu verileri satır satır silerek transaction-log'a dokunmuyor

Şimdi tekrar Departman tablomuzdan DepartmanAdi “finans” olan satırı silelim.

Tablodan Belli Bir Şarta Uygun Olan Veriyi Silmek

```
delete from Departman
where DepartmanAdi = 'Finans'
```

Result penceresini incelediğimizde X row Affected mesajını görüyoruz. Bu mesaj sorgumuzun sonucunda etkilenen satır sayısını söyler.

Update Komutu

Update komutu tablodaki var olan bir verinin değiştirilmesi için kullanılır. Yine tüm tabloyu etkileyen bir komut olduğu için delete gibi dikkatli kullanılması gerekir.

Tüm satırları etkileyen update örneği

Tablodaki Kayıtları Güncellemek

```
update Calisanlar
set Maas = Maas + (Maas / 100) * 10
```

Yukarıdaki sorgu da Calisanlar tablosundaki Maas alanını güncelledik. Set ile değiştirilecek alana değer atıyoruz. Yukarıdaki örnekte çalışanlarımıza maaşları üzerinden %10 zam yaptık.

Şimdi tek bir satırı değiştirelim

Tablodaki Tek Bir Kaydı Güncellemek

```
update Calisanlar
set Maas = Maas + (Maas / 100) * 10, DepartmanID = 2
where CalisanID = 1
```

Yukarıdaki örnekte CalisanIDsi 1 olan çalışanın Maaşına %10 zam yaptık ve DepartmanID 'sini 2 olarak değiştirdik. Eğer birden fazla sütun değeri değişecekse set birkere yazılıp sütunlar ' , ' ile ayrılır. Yine önemli bir hatırlatma olarak update ve delete komutlarını where ile kullanmaya dikkat edelim. Ve bu tür işlemlerde eğer verileriniz önemliyse kesinlikle veritabanının yedeğini alalım.

JOINS

Join ifadesi iki veya daha fazla tabloyu result penceresinde birleştirmek için kullanılır. Bu farklı tablolar genellikle fiziki veya mantıksal ilişkilendirilen tablolardır. Join; ilişkisel veritabanlarında bir tablodaki ikincil anahtar (foreign key) üzerinden parent tablodaki bütün sütunlara erişmek için kullanılır. Yani birleştirme benzer alanlar üzerinden yapılır

Dikkat : Join ile tablolar fiziki birleştirilmezler sadece sorgu sonucunda birleştirilerek daha anlamlı sonuçlar elde etmek için kullanılır

T-sql 4 farklı join türü destekler. Bunlar;

- Inner Join
- Outer Join
- Self Join
- Cross Join

Inner Join

Inner join (iç birleştirme) iki tablodaki kesişen kayıtları elde etmek için kullanılır. Genelde en çok tercih edilen join yöntemidir.

Inner Join

```
select
    c.Adı,
    c.SoyAdı,
    c.DepartmentID as ÇalışanDepartmentID,
    d.DepartmentID,
    d.DepartmentAdı
from Çalışanlar as c
inner join Departman as d
on c.DepartmentID = d.DepartmentID
```

Yukarıdaki örnekte Çalışanlar ve Departman Tablolarını departmentID'leri üzerinden birleştirdik. Select ifadesinden sonra sütun isimleri yerine '*' ile her iki tablonun bütün sütunlarını getirebilirdik. Sorguyu çalıştırıp result sette sonucu incelediğimizde Çalışanların DepartmentID alanı ile Departmanların DepartmentID alanlarını karşılaştırırsak joini daha iyi anlarız. Sadece ilişkili kayıtların geldiğini ve iki alan değerlerinde birbirine eşit olduğunu görürüz.

Inner Join Kullanımı

```
select
    c.*,d.DepartmentAdı
from Çalışanlar as c
inner join Departman as d
on c.DepartmentID = d.DepartmentID
```

Yukarıdaki örneğimizde **c.*** ifadesi çalışanlar tablosunun bütün sütunlarını, departman tablosunun ise sadece departmentAdı alanını getirmemizi sağlar.

Inner join ile iki tablo result penceresinde artık tek bir tablo gibi kullanılabilir. Yani istediğiniz sütunları getirir ve filtreleme, grüplama veya sıralama işlemleri yapabilirsiniz.

Join Örneği

```
select
    CONCAT(c.Adı, ' ', c.SoyAdı) as [Full Adı],
    d.DepartmentAdı
from Çalışanlar as c
inner join Departman as d
on c.DepartmentID = d.DepartmentID
```

Yukarıdaki sonucu incelediğimizde ise çalışanların adlarını ve departman isimlerini listeleterek sonucun daha anlamlı olmasını sağlıyoruz.

Aşağıdaki sorgumuzda sonucu DepartmanAdına göre grüplayarak, grüpl küme elemanlarını saydırıyoruz ve hangi departmanda kaç çalışmamız varmış görebiliyoruz

Join Örneği ve Aggregate Function Kullanımı

```
select
    d.DepartmanAdi,
    COUNT(*) as CalisanSayisi
from Calisanlar as c
join Departman as d
on c.DepartmanID = d.DepartmanID
group by d.DepartmanAdi
```

Sizlerde aynı yöntemi kullanarak, Departmanlara göre maaş toplamalarını, Departman bazlı min ve max maaşları listeleyiniz.

Son join örneğimizde inner join yerine join yazdık. T-sql de join ve inner join ifadeleri aynı anlama geliyor. İstedığınızı yazabilirsiniz.

Joini iki farklı şekilde yazabiliriz. İlkini zaten yazdık. İkinci tablo birleştirme yöntemi örneği aşağıdadır;

Join Kullanımı

```
select
    *
from Calisanlar as c,Departman as d
where c.DepartmanID = d.DepartmanID
```

Yukarıdaki sorgumuz aslında SQL-ANSI- 89 Syntax standardıdır. Yani daha eski bir birleştirme yöntemidir. Farklılıklarına baktığımızda Tablolarımızı join yazarak birşeltirmek yerine from komutundan sonra ' , ' (virgül) ile ayırıyoruz ve where ifadesi ile on komutunun görevini yapıyoruz. İki arasında performans farklılığı yoktur. Bu yöntem ansi-89 sözdizimi, join yöntemi ise ansi-92 sözdimi ile yazılır. T-sql her ikisini destekler. Fakat tavsiyem ansi-92 ile yazmanızdır. Çünkü yazım kuralı vardır. Eğer joinde on komutunu kullanmazsanız syntax hatası alırsınız. Ansi-89 ile yazarsanız where yazmasanız dahi çalışır ve tutarsız bir tablo ile karşılaşabilirsiniz.

Outer Joins

Outer joinler (dış birleşim) Left, Right, full join olarak 3'e ayrılır. Outer joinler ansi-92 ile desteklenmiştir. Inner join gibi yazım alternatifi yoktur. Sadece outer yazma zorunluluğunuz yoktur. Sırasıyla inceleyelim

Left Outer Join

Left outer veya left join iki tablodaki birleştirirken öncelikle sol tablonun tamamı sağ tablonun sadece kesişenlerini alır. Sol tablodaki ve sağ tablodaki kesişen kayıtlar ve sol tablodaki kesişmeyen kayıtlar listelenir. Sol taraftan kesişmeyen kayıtların karşısında NULL değerler vardır

Outer Join (Left) Kullanımı

```
select
    *
from Departman as c
left join Calisanlar as d
on c.DepartmanID = d.DepartmanID
```

Sorgu sonucunu incelediğimizde departman tablosu sol tablo, calisanlar tablo sağ tablodur. Ve sol tablada ilk ve Teknik Çizim alanlarının karşı satırlarının NULL olduğunu görürüz. Inner joinde farkıda burada ortaya çıkıyor zaten.

Left join ile tekrar bir örnek yapalım ve Çalışanları olmayan Departmanları bulalım

Outer Join Örneği

```
select
    *
from Departman as d
left join Calisanlar as c
on c.DepartmentID = d.DepartmentID
where c.DepartmentID is null
```

Sonucu incelediğimizde sadece İK ve Teknik Çizim satırlarının geldiğini ve sağ taraftaki bütün sütunların NULL olduğunu göreceğiz. Bu alanlardan hepsi NULL olduğu için herhangi birisini filtrelemeye dahil etmemiz bize istediğimiz sonucu verecektir. Gördüğünüz gibi hem birleştirme işlemi yaptık hemde iki tablo içerisinde olan ve olmayan kayıtları analiz ettik.

Şimdi aşağıdaki sorguyu dikkatlice okumanızı, ve daha sonra yazarak çalıştırmanızı tavsiye ediyorum.

Outer Join Örneği

```
select
    d.DepartmentAdi,
    COUNT(c.DepartmentID) as CalisanSayisi,
    COUNT(*) as CalisanSayisi
from Departman as d
left join Calisanlar as c
on c.DepartmentID = d.DepartmentID
where c.DepartmentID is null
group by d.DepartmentAdi
```

Sıra Sizde

Countların neden farklı geldiğini yorumlayınız.

Right Outer Join

Left joinin tam tersidir.

Outer Join (Right) Örneği

```
select
    *
from Calisanlar as c
right join Departman as d
on c.DepartmentID = d.DepartmentID
```

Full Outer Join

Her iki tablodaki kesişen ve kesişmeyen kayıtları elde etmek için kullanılır

Full Outer Join Örneği

```
select
    *
from Calisanlar as c
full join Departman as d
on c.DepartmanID = d.DepartmanID
```

Self Join

Tabloların kendisiyle birleştirilmesi işlemidir. Örnek olarak Kategoriler tablosu oluşturalım.

Self Join Örneği

```
create table Kategoriler
(
    ID int identity(1,1),
    Adi nvarchar(50),
    UstKatID int
    constraint pkKategoriler primary key(ID),
    constraint fkKategoriler foreign key(UstKatID) references Kategoriler(ID) -- UstKat foreign key olarak tanımladık
    ve Aynı tablo üzerinde ID alanı ile bire-çok ilişki kurduk
)
```

Yukarıdaki örnekte Aynı tablo içerisinde alt kategori ve üst kategori bilgilerini tutuyoruz. Bu sayede sınırsız kategori ekliyoruz. Üst kategori olan satırların UstKatID'lerini null ekliyoruz. Alt kategori olanların ise UstKatID'sine aynı tablodaki UstKatID alanının ID alanını set ediyoruz. Örnek olarak bir kaç veri ekleyelim.

Insert İşlemi

```
insert Kategoriler
(ID,Adi,UstKatID)
values
(1,'Ses Görüntü',Null), -- Üst kategori
(2,'Telefon',Null), -- Üst kategori
(3,'Televizyon',1), -- Ses Görüntü kategorisinin alt kategorisi
(4,'Projeksiyon',1), -- Ses Görüntü kategorisinin alt kategorisi
(5,'Hafıza Kartı',2), -- Telefon kategorisinin alt kategorisi
(6,'Kılıf',2), -- Telefon kategorisinin alt kategorisi
(7,'Cepli Kılıf',6), -- Kılıf kategorisinin alt kategorisi
(8,'Cepsiz Kılıf',6), -- Kılıf kategorisinin alt kategorisi
(9,'Spor',NULL) -- Üst kategori
```

Kategorilerimize yukarıdaki örnekte olduğu gibi ekliyoruz. UstKatID alanı NULL olanlar bizim parent kategoridir. Null olmayanlar ise parent kategorilerinin alt kategorileridir. Değerlerini ID alanı ile karşılaştırabilirsiniz.

Şimdi bir tane self join örneği yapalım

Self Join Örneği

```
select * from Kategoriler as ustKat
inner join Kategoriler altKat
on ustKat.ID = altKat.UstKatID
```

Sorguyu çalıştırıp sonucu incelediğimizde birbirine bağlı olan verilerin aynı satırda olduğunu görüyoruz.

Cross Join

Her bir satırın iki tablodaki değerler ile eşleştirilmesini sağlar

Cross Join Örneği

```
select CalisanID,d.DepartmanID from Calisanlar as c
cross join Departman as d
```

Yukarıdaki örnekte her bir çalışanın CalisanID ile DepartmanID'nin eşleştirildiğini görüyoruz.

Derived Table

Türetilmiş tablolar bir veya daha fazla tablonun birleştirilerek select sonucu oluşan sanal tablolardır. Derived table sadece anlık çözüm üretmek için kullanılır. Derived table from komutundan sonra içteki select sorgusunun sonucunu miras alır ve döndür. Birkaç örnek ile inceleyelim

Derived Table Örneği

```
select * from( -- table start
select * from Calisanlar -- iç sorgu satırları dış sorguya aktarılır
) as derivedTable -- table end
```

Yukarıdaki örneğimizde Calisanlar tablosundaki kayıtları dış sorguya aktardık ve dış sorgu sonucunda oluşacak tabloya derivedTable ismini verdik.

Dikkat : Derived table kalıcı olarak saklana bir tablo değildir. As deyiminden sonra tek satırlık kullanım alanı vardır (where order by group by kullanılabilir)

Derived Table Örneği

```
select DepartmanAdi,CalisanSayisi
from(
    select
        DepartmanID,
        COUNT(*) as CalisanSayisi
    from Calisanlar
    group by DepartmanID
) as DepartmanOzet
inner join Departman as dp
on DepartmanOzet.DepartmanID = dp.DepartmanID
```

Örneğimizde iç sorgudan Calisanlar tablosunun DepartmanID alanına göre grupladık ve Count ile alt kümelerini saydırdık. Daha sonra dış sorguya aktarılan veriyi Departman tablosuyla birleştirerek DepartmanAdi ve çalışanSayılarını listeledik.

Anlık (Geçici) bir tabloya neden ihtiyaç olduğunu aşağıdaki sorguda tekrar inceleyelim.

Örnek Kullanım

```
select
    CONCAT(Adi, ' ', SoyAdi) as FullAdi
from Calisanlar
where FullAdi = 'Ekrem Yıldırım'
```

Msg 207, Level 16, State 1, Line 454

Invalid column name 'FullAdi'.

Bu örneğimizde aliases kullandığımız FullAdi bölümünü where ile kullanamıyoruz.

Dikkat : Aliases kullanılan alanlar where ile filtrenemez. Sadece order by ile sıralama işlemine dahil edilebilirler.

Şimdi aynı sorguyu derived table olarak yazalım ve çalıştıralım.

Örnek Kullanım

```
select * from (
select
    CONCAT(Adi, ' ', SoyAdi) as FullAdi
from Calisanlar
) as geciciTablo
where geciciTablo.FullAdi = 'Ekrem Yıldırım'
```

Örneği incelediğimizde Calisanlar tablosundaki satır ve seçilen sütunu geciciTablo olarak isimlendirdiğimiz derived table aktardık. Daha bu tabonun FullAdi alanı üzerinden filtreleme yaptık.

Sub Query

T-Sql ile sorgu içerisinde sorgular yazabilirsiniz. Bazı durumlarda filtreleme yapmak için veya select ifadesinde başka bir tablodan sütun çıkartmak için kullanılır.

İlk örneğimizi yapalım.

Örnek Kodlar

```
declare @MaxMaas decimal

select @MaxMaas = MAX(Maas) from Calisanlar

select * from Calisanlar
where Maas = @MaxMaas
```

Yukarıdaki örneğimizde Max fonksiyonu ile Calisanlar tablosundaki maximum maaş alan çalışanımızın maaş bilgisini @MaxMaas değişkenine atadık ve altındaki select ifade ise filtreleme yaparak en çok maaş alan çalışanımızın bilgilerine erişti. Bu sorgu bu şekilde uzatmak yerine subquery ile de yazabilirdik.

SubQuery Örneği

```
select *  
from Calisanlar -- dış sorgu  
where Maas = (  
    select Max(Maas) from Calisanlar  
    ) -- iç sorgu
```

Örnekte iç sorgudan çıkan değer, dış sorguya aktarılır ve ilk örneğimizdeki sonucu elde etmiş oluruz. Bu şekilde aynı tablo veya farklı tablolar üzerinde çeşitli filtrelemeler yapabiliriz.

SubQuery ile Filtreleme Örneği

```
select * from Departman  
where DepartmanID not in (select DepartmanID from Calisanlar)
```

Bu örneğimizde iç sorguda Calisanlar tablosundan DepartmanID leri çıkardık ve DepartmanID tablosunun departmanID alanlarını karşılaştırdık. Not in komutu ile Eşit olmayanlar dedik ve Çalışanı olmayan departmanları elde etmiş olduk.

Correlated Sub Query

İlişkisel alt sorgular, SubQuery olarak yazılan iç sorgunun, dış sorgunun her bir satırı için çalışmasını ifade eder.

İlişkisel Alt Sorgu Örneği

```
select
    Adi,
    SoyAdi,
    (select DepartmanAdi from Departman as d
     where d.DepartmanID= c.DepartmanID) as Departmani
from Calisanlar as c
```

Yukarıdaki örneğimizde select ifadesi içerisindeki sorgu dış sorgudan gönderilen parametre ile çalıştırılır ve geriye bir değer döner. Yani dış sorgudan DepartmanID alanı input iç sorgudan DepartmanAdi output olur.

Sql Pivot Sorguları

Pivot sorgular bir select ifadesiyle satır olarak dönen sonuçların sütunlara çevrilmesi için kullanılır. Örneğimizde ilk olarak Departmanlar ve Maliyetlerini listeliyoruz

Departman ve Maliyet Listeleme Örneği

```
select
    DepartmanAdi,
    Sum (c.Maas) as ToplamMaliyet
from Departman as d
inner join Calisanlar as c
on d.DepartmanID = c.DepartmanID
group by DepartmanAdi
```

Yukarıdaki sorguda sonuç satır olark listelendi. Şimdi bu sorguyu bir derived table içerisine alıyoruz ve pivot yapıyoruz.

Pivot Table Örneği

```
select * from ( -- derived tablo başlangıcı
select -- iç sorgu başlangıcı
    DepartmanAdi,
    Sum (c.Maas) as ToplamMaliyet
from Departman as d
inner join Calisanlar as c
on d.DepartmanID = c.DepartmanID
group by DepartmanAdi -- iç sorgu bitişi
) as pivotTablo -- derived tablo bitişi
```

```
pivot ( -- pivot table başlangıcı  
      SUM(ToplamMaliyet)  
      for DepartmanAdi in ([Grafik],[Yazılım],[Muhasebe])  
    ) as pivotTable -- pivot table bitişi
```

Sonucu incelediğimizde listelenen sonucunun satırdan sütun'a çevrildiğini görüyoruz.

INDEX MİMARİSİ

Indexler veritabanımızın ön önemli performans nesneleridir. Bir veritabanının görevi sadece veri saklamak değildir. Aynı zamanda depolanan veriye hızlı erişimde sağlamalıdır. Sql Serverdaki Index yapısı bu performans görevini görür. Peki neden indexlere gerek vardır sorusuna gelince bir tabloya select ifadesi uyguladığımızda veriye erişim için **TABLE SCAN** yapılır. Table Scan yapılması demek dataların tek tek okunması demek. Yada bir filtreleme işlemi yaptığımızda yine aynı şekilde table scan yapılarak datalar satır satır okunur ve işlem data bulunana kadar devam eder. İşte bu durum veritabanlarında istenmeyen bir durumdur ve yazılan sorgunun cevabının geç gelmesine neden olur. Bir kitap içerisinde bir konuyu aradığımızda index bölümüne bakarız ve konu başlığına bulunduğu sayfayı bularak konuyu buluruz. 100 sayfalık bir kitabın tüm sayfalarını taramaktansa sadece 3 sayfa tarayarak sonuca erişiyoruz. İşte sql server index yapısını kullandığımızda aynen bu şekilde işlem yapıyor. Eğer kitap içerisinde index olmasaydı konuyu bulana kadar kitabın tüm sayfaları incelenenecekti ve zaman kaybına neden olacaktı. İşte sql server index kullandığımızda dataların incelemek yerine sıraladığı dataların sadece indexine bakarak ilgili sayfayı tarar ve veriye erişir. Sql server da bu tarama işlemine Index Scan denir. Filtreleme işlemi yaptığımızda da Index Seek (Arama) yapar. Sql Server da indexleri ColumnStore index ve RowStore index olarak inceliyoruz. ColumnStore indexler yeni nesil bir index türüdür. RowStore indexler ise daha eski ve kullanışlı index türleridir.Şimdi sırasıyla Bu index türlerini inceleyelim.

- Clustered Index (Row)
- Non-Clustered Index (Row)
- Column Store Index
- Non-Clustered Column Store Index

Clustered Index

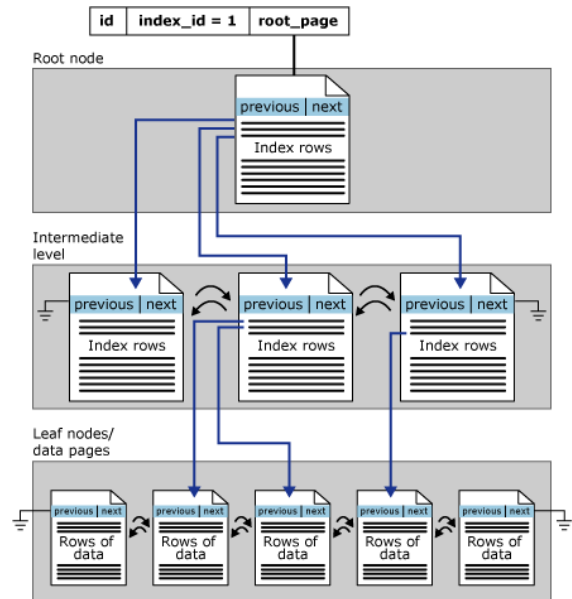
Fiziksel Index olarakta bilinir. Fiziksel denmesinin neden sql indexleme algoritmasını **B-Tree(Blanced-Tree)** olarak bilinen yapıda tutar. Bu b-tree yi inceliyor olacağız. Ancak Clustered denilmesindeki neden bu ağaç yapısında verinin tamamen kendisinin tutulmasıdır. Aşağıdaki resimde b-tree yapısını inceleyelim.

B-Tree 3 ana yapıdan oluşur;

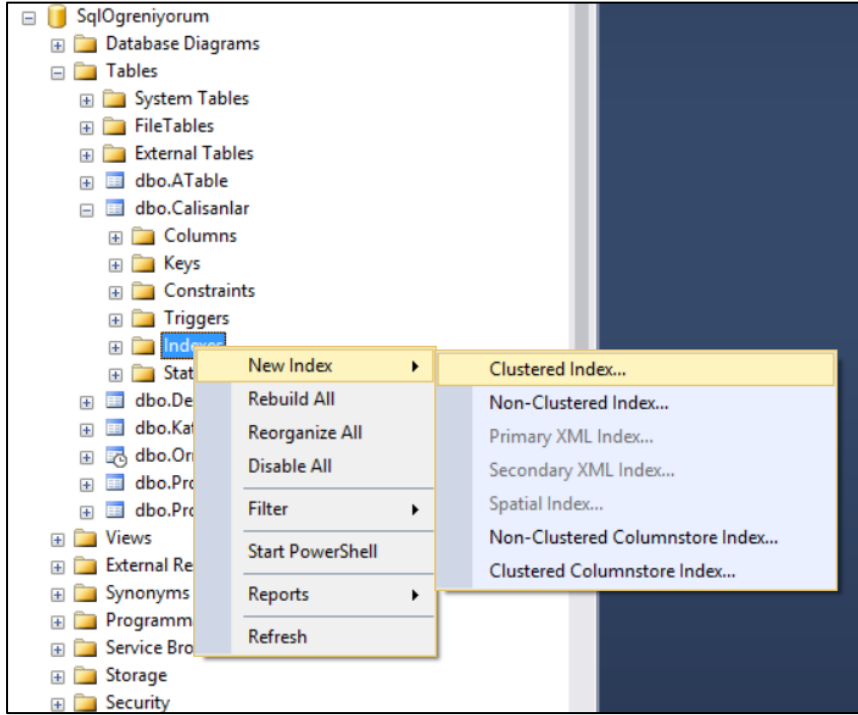
Root Level : B-tree nin en üst seviyesidir. B-tree bu seviyede dallanmaya başlar. Bu seviyenin hemen altında;

Intermediate Level : Root level datanın büyüklüğüne göre Intermediate Level'i işaret eder. Intermediate level verileri işaret eden seviyedir. 100 satırlık bir tabloda iki adet Intemediate level olduğunu düşünecek olursak verileri 50'şer şekilde 2 sayfada tutulur.

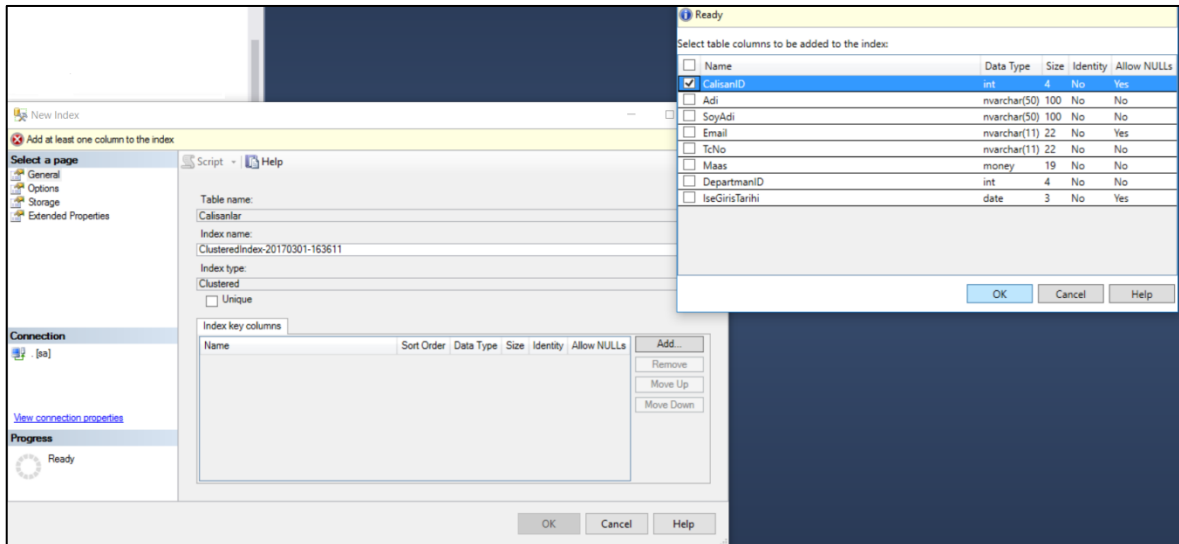
Leaf Level : Leaf level yada data page olarak bilinir bu seviyede Intermediate Levelin işaret ettiği veriler (datanın kendisi) bulunur.



Sql Server bu yapıda dataları sıralı bir şekilde oluşturur ve bütün sorgular ve filtrelemeler bu yapı üzerinden devam eder. Clustered Indexler tabloda sadece 1 adet tanımlanabilir. Bunun nedeni Leaf levde datanın tüm satırları ile tutulmasıdır. Tablo oluştururken oluşturduğumuz her index sql server tarafından Clustered Index olarak tanımlanır. Ama bir primary keyimiz yoksa veritabanımızdaki table sekmesi altından tablomuzun Indexes sekmesinden sağ click new Index sekmesinden clustered Indexi seçeriz.



Daha sonra karşımıza gelen new index penceresinden Add düğmesi ile Select Column Penceresinden Indexlenecek sütunu seçelim ve ok düğmesiyle seçimi onaylayalım.



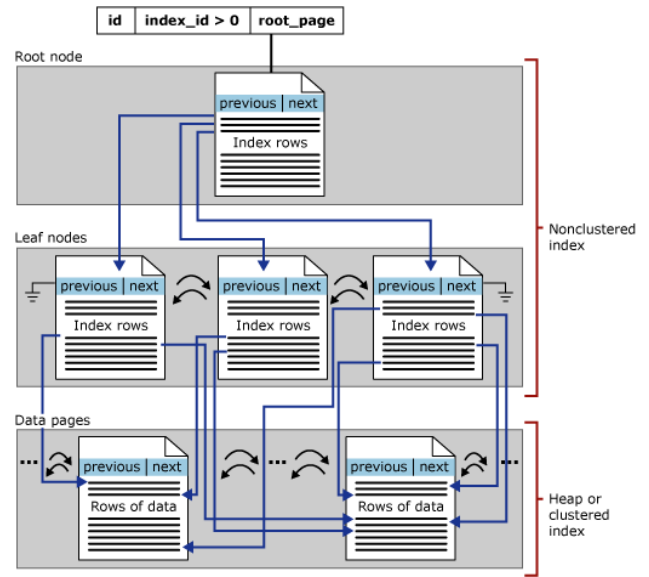
Burada dikkat etmemiz gereken unsurlardan biriside Index olarak belirlenecek alanımızın numeric olmasıdır.

Non-Clustered Index

Non-Clustered Index, sql serverda bir tabloda birden fazla index oluşturmak için kullanılan bir index türüdür. Çünkü Clustered Index 1 adet tanımlanabiliyordu. Aradaki fark sadece tanımlanma sayısı değil tabiki. Non-Clustered Index

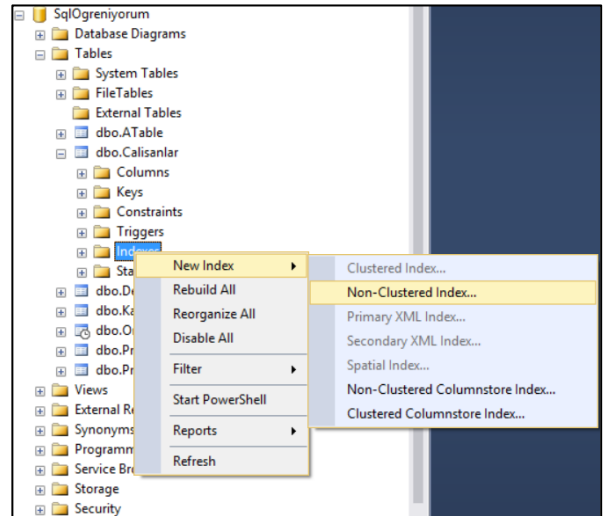
ile birlikte b-tree yapısıda değişiyor. B-tree yapısında Non-Clustered Index kullandığımızda daha önce bahsettiğimiz leaf levelde verinin kendisi yerine sadece indexlenen sütun veya sütunlar tutulur. Diğer sütunlar ise heapta tutulur. Ayrıca leaf levelde datayı işaret eden pointerlar bulunur. Dataların kendisi heap denilen bölgede bulunurlar. Non-Clustered Index özellikle primary key dışında kalan ve sık filtreleme yapılan alanlar için tanımlanır. En güzel örneğininde online giriş sisteminin verilerinin tutan bir kullanıcılar tablosunda kullanıcıadı ve şifre alanlarını çok sık bir şekilde kullanırız. Bu durumda bu iki alan non-clustered index olarak tanımlanabilir ve sadece bu iki veriye erişim istenildiğinde hızlı bir şekilde değerleri elde edebiliriz. Ancak Non-Clustered Index bu iki alana erişim için performans kazanılmasına neden olurken bu iki alan üzerinden diğer kolonları çağırıcaksak performans kaybına neden olur. Bunun nedeni dataların leaf levelde değil heapta olmasıdır. Bu iki alan üzerinden datalara pointer (row locator) ile erişim yapılır, bu işleme key lookup denir. Key lookup performans kaybına neden olan bir işlem olduğu için kullanıcıadı ve şifre sütunlarıyla erişilecek alanlar included column olarak ayarlanırsa key lookup işlemi yapılmaz. Kısacası dikkatli kullanmakta fayda vardır. Şimdi Non-clustered index yapısındaki B-Tree'yi inceleyelim.

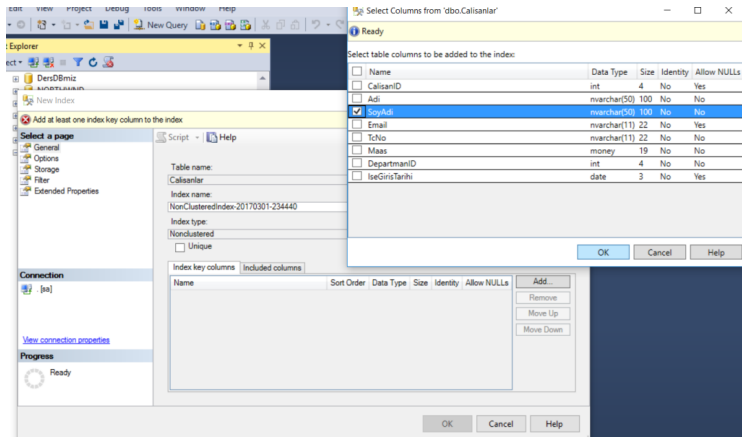
Yandaki resmi incelediğimizde yine bir adet root level onun hemen altında ise leaf level var. Leaf levelde sadece non-clustured index olarak tanımlı sütunlar ve heaptaki veriyi işaret eden row locatorler mevcut. Örneğin bir select ifadesinde where komutu ile non-clustered index alanı üzerinde bir filtreleme yaptığımızda veriye erişim için için leaf levelden işaret edilen sütunlar data pages olan heap bölgesinden alınır. Bu işleme look up işlemi demiştik. Özellikle non-clustered index üzerinde include column özelliği kullanılarak leaf levele o sütunlarda dahil edilirler ve look up işleminin önüne geçilir.



Örnek olarak çalışanlar tablomuza bir non-clustered index tanımlayalım. Daha önce clustered indexde izlediğimiz adımlarını izliyoruz.

Dikkat ederseniz Clustured Index alanı pasif. Bunun neden daha önce clustered index oluşturmuştuk. Zaten bir adet clustered index oluşturabildiğimiz için Non-Clustered index oluşturuyoruz. Non-Clustered index seçeneğini click yapalım ve karşımıza gelen pencereyi alttaki resimde inceleyelim.





Non-Clustered index seçeneğini click yaptığımızda new index penceresinden add butonu ile select column penceresinden SoyAdi alanını seçerek Ok düğümüyle onaylıyoruz.

Şimdi Departman ve Çalışanlar tablomuzdaki index yapısını aşağıdaki sorgu ile inceleyelim.

```
select * from sys.dm_db_index_physical_stats(Db_ID('SqlOgreniyorum'),Object_ID('Deparman'),-1,null,'Detailed')
```

```
select * from sys.dm_db_index_physical_stats(DB_ID('SqlOgreniyorum'),OBJECT_ID('Calisanlar'),2,null,'detailed')
```

Sorgu sonucu dönen tabloudaki kolonları incelediğimizde

- **databaseid** = veritabanı idsi
- **objectid** = tablo idsi
- **indexid** = index idsi
- **index_type_desc** = index tip açıklaması
- **index_depth** = index derinliği
- **index_level** = level seviyesi (0 leaf level, 1 intermediate level, 2 root level)
- **page_count** = level'deki sayfa sayısı
- **record_count** = sayfalardaki kayıt sayıları
- **avg_fregmantation_in_percent** = indexlerdeki fregmantasyon (bozulma) oranıdır.

Index Fragmentation (Parçalanma)

Indexlerin bozulmasıdır. Tablomuz insert, update, delete işlemi gördükçe indexler fregmante olurlar. Indexin fregmante olması performansı olumsuz yönde etkiler. Bu durumu engellemek için indexleri defragmente etmemiz gerekir.

Index Defragmentation (Birleştirme)

Bozulan indexlerin düzeltilme işlemidir. Yani index bakımıdır. Bu bakımların yapılabilmesi için indexin bozulma oranını belirlemek gerekir.

Aşağıdaki sorguyu çalıştırdığımızda indexin bozulma oranı belirlenir.

```
select avg_fragmentation_in_percent from sys.dm_db_index_physical_stats(Db_ID('SqlOgreniyorum'),Object_ID('Calisanlar'),-1,null,'Detailed')
```

Şimdi index bozulma oranına göre bakım sürecini inceleyelim

avg_fragmentation_in_percent	İşlem
> 5% and < = 30%	Reorganize
> 30%	Rebuild

Bozulma oranlarına göre düzenleme işlemi yukarıda belirtilmiştir. Bir Indexi Reorganize etmek leaf levelde ki page'leri tekrar sıralamaktır. Rebuild işlemiyse indexlerin tekrar drop-create edilmesidir. Indexleri düzenlemek için tablomuzun indexes sekmesinde Rebuild ve Reorganize seçenekleri mevcuttur.

Clustered Column Store Index

Yeni gelen bir index türüdür. Column Store Index clustered ve non-clustered indexlerin row bazlı depolamasının aksine verileri column'larda tutar. Column Store indexlerde veriler segment adı verilen kolonlar halinde tutulur. Column Store Index oluşturmadan önce SqlOgreniyorum veritabanına 1 adet IndexTable oluşturalım

```
create table Indextablo
(
    Col1 int identity(1,1),
    Col2 nvarchar(50),
    Col3 nvarchar(50)

    constraint pkIndexTablo primary key(Col1)
)
```

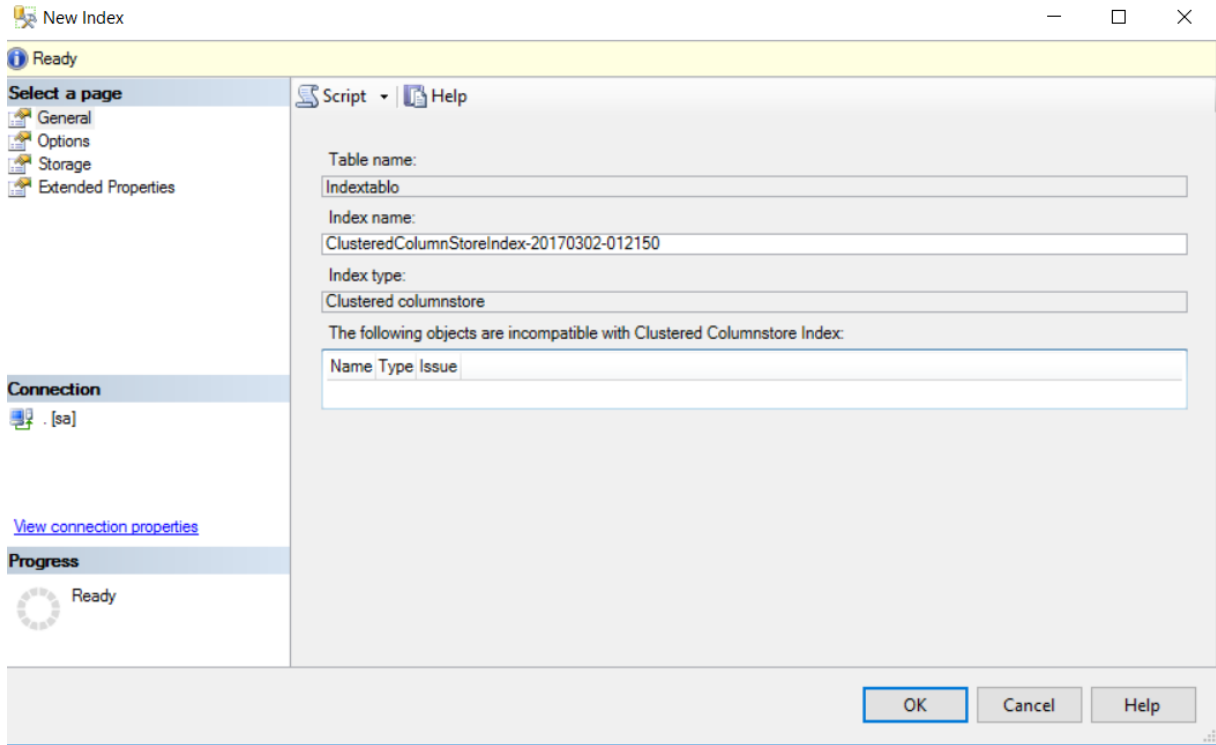
Daha sonra bir döngü yardımıyla bu tabloya 500 adet veri girelim.

Döngü ile Veri Ekleme

```
declare @sayac int = 0

while @sayac < 500
begin
    insert Indextablo
    (Col2,Col3)
    values
    ('A' + Convert(nvarchar,@sayac),'B' + Convert(nvarchar, @sayac))
    set @sayac = @sayac + 1
end
```

Şimdi ilk olarak Indextablo sekmesi içerisindeki indexes sekmesinden olan indexi silmemiz gerekiyor. Indexe sağ tıklayarak indexi siliyoruz. Daha sonra tekrar aynı sekmede new index > Column Store indexe click yapıyoruz.



Yukarıdaki pencerede dikkat ederseniz add butonu bulunmuyor. Bunun nedeni ColumStore index oluştururken bütün sütunlar indexe dahil edilirler. Ok düğmesiyle onayladıktan sonra indeximiz oluşacaktır. Dilerseniz Index name alanından anlaşılabilir bir isim verebilirsiniz.

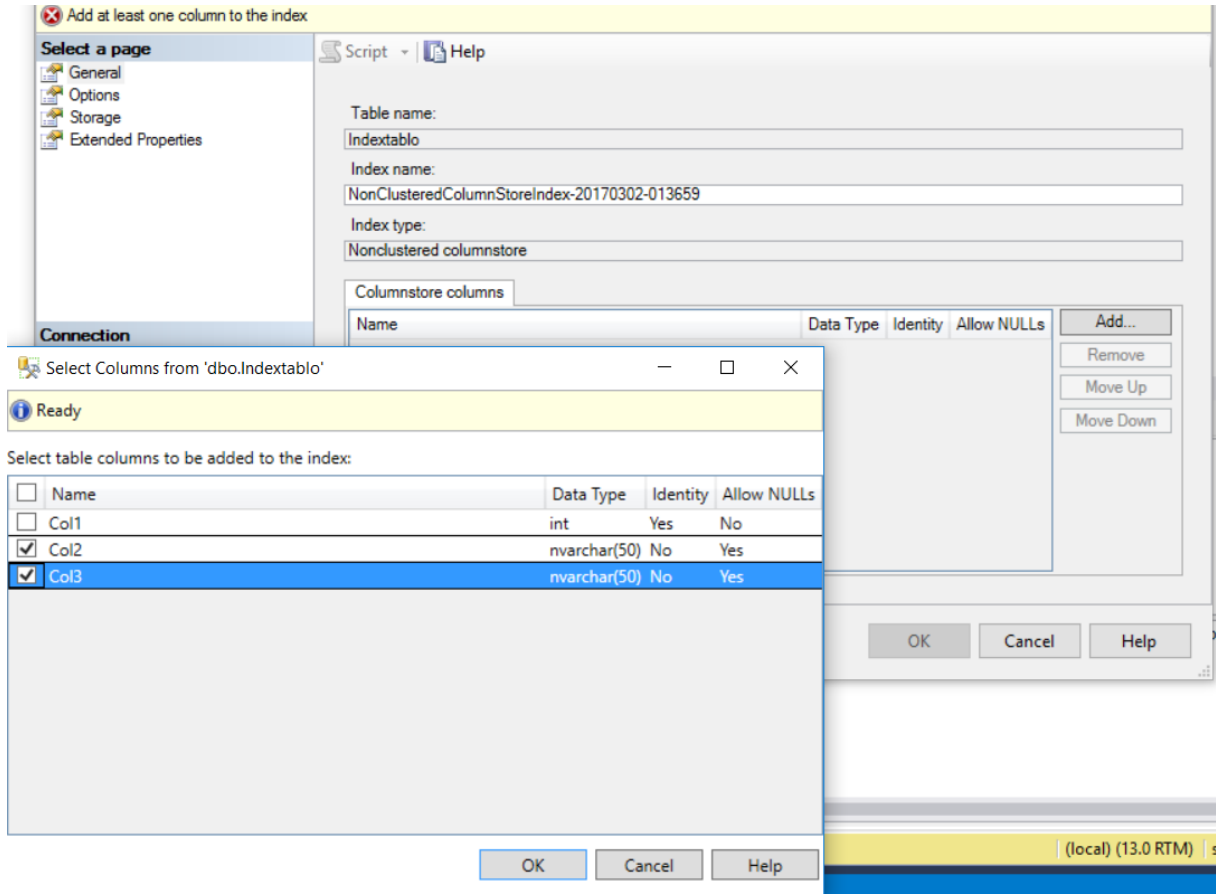
ColumnStore indexlerin detaylarını aşağıdaki sorgu sonuçlarından inceleyebiliriz.

ColumnStore Index

```
select * from sys.column_store_row_groups
select * from sys.column_store_segments
```

Non-Clustered Column Store Index

Non-Clustered Column Store index bir diğer column store index türüdür. Bu index türünde diğer indexe göre tablonun tamamını indexlemek yerine sadece istediğimiz sütunları indexleyebiliriz. Öncelikle non-clustered column store index oluşturmak için daha önce oluşturduğumuz column store indexi silelim. Bunun nedeni Clustered Index olan bir tabloda non-clustered column store index tanımlanamaz. Daha önce oluşturduğumuz IndexTablo içerisindeki indexi silelim. Daha sonra aynı sekmeden new index > Non-Clustered Column Store Index seçeneğini seçelim ve aşağıdaki pencereyi inceleyelim.



Bu pencere add butonu ile gelen pencereden Col2, ve Col3 sütunları seçip Ok düğmesine tıklıyorum ve diğer pencereden de Ok düğmesine tıklayarak indeximizi oluşturuyoruz. Bu pencerede dikkat etmemiz gereken diğer bir nokta ise sütunları dikkatli seçmeliyiz çünkü bir tabloda en fazla bir adet columnstore index tanımlanabilir. Yine index detayını aşağıdaki sorgumuzla görebiliriz

NonColumnStore Index

```
select * from sys.column_store_row_groups
select * from sys.column_store_segments
```

Her iki columnstore index yapısında da veriler sütunlar halinde segment denilen yapıda sıkıştırılarak saklanırlar. Ve her 1 milyon satır gruplara ayrılır. Bu index türlerinde sorgulama yaptığımız zaman veri satırlarda aranmaz sadece sütunlarda aranır. Yine segmentler row gruplara bölündüğü için basit düşünecek olursak 1 milyon 100 bin kaydın olduğu bir tabloda satır sayısı 1 milyonu geçtiği için 2 row group oluşur ve 1 milyon 101.ci kayıt arandığında ilk row group direk atlanır ve ikinci row group içerisindeki segmentlerde arama işlemi yapılır. Buda filtreleme işlemlerinde sorgu performansı arttıran bir diğer önemli özelliğidir.

VIEWS

View sql server ortamında sanal tablo olarak kullanılan gerçek tablodan veriyi alarak özetleyen tablolardır. Daha önce bahsetmiş olduğum derived tabloların aksine anlık değil kalıcı olarak saklanan sql server objeleridir. Viewlar sonraki aşamalarda da kullanılabilir. Gerçek tablolardan farklı select sonucu oluşurlar. Sürekli yazdığımız sql sorgularımızı veya karmaşık iç içe yazılan (subquery) veya çoklu tablolarımızın (joinler) ihtiyaç halinde sürekli yazmak yerine birkere yazıp sürekli kullanma kolaylığı sağlar.

Şimdi SqlOgreniyorum veritabanına çalışanlar tablomuzdan ad,soyad,maas alanlarını getiren bir view oluşturuyoruz

View Örneği

```
use SqlOgreniyorum
go
create view CalisanOzetBilgi
as
select Adi,SoyAdi,Maas from Calisanlar
```

Yukarıdaki sorguyu çalıştırdığımızda select sonucu oluşan 3 sütunlu bir CalisanOzetBilgi viewı oluşturduk. Şimdi bu oluşturduğumuz viewı çağıralım

View Çağırma

```
select * from CalisanOzetBilgi
```

Sonucu incelediğimizde view içerisindeki sorgumuzdan sonucun döndüğünü görüyoruz. Bunu bir metot gibi düşünecek olursak her view çağırıldığında gövdesindeki komut çalışacak ve sonuç listelenecek. Aslında sonuç değişmesede bir sonraki kullanımlarda sadece view ismini çağıracağız.

Dikkat : View oluştururken kesinle select ifadeleri içerisindeki sütun isimleri belli olmalıdır(İsimsiz sütun olmamalıdır). Ve ayrıca sütun isimleri farklı olmalıdır

Şimdi yine bir view yazalım ve Çalışanlarımızın Ad,Soyad ve Departman İsimlerini listeleyelim. View içerisinde bu sefer tek tablo yerine iki tablo kullanacağız ve view ile sonucu tek tablo olarak listeleyeceğiz

View Kullanım Örneği

```
create view vwCalisanlarveDepartmanlari
as
    select * from Calisanlar as c
    inner join Departman as d
    on c.DepartmentID = d.DepartmentID
```

Msg 4506, Level 16, State 1, Procedure vwCalisanlarveDepartmanlari, Line 3 [Batch Start Line 18]

Column names in each view or function must be unique. Column name 'DepartmentID' in view or function 'vwCalisanlarveDepartmanlari' is specified more than once.

Sorgumuzu çalıştırdığımızda aldığımız hata her iki tabloda da DepartmentID alanlarının olmasından kaynaklıdır. Daha önce view içerisindeki sütun isimlerinin benzersi olması gerektiğinden bahsetmiştik.Sorgumuzu aşağıdaki gibi tekrar güncelleyerek çalıştırıyoruz.

View Örneği

```
create view vwCalisanlarveDepartmanlari
as
    select
```

```
c.Adi,  
c.SoyAdi,  
d.DepartmanAdi,  
c.DepartmanID as CalisaninDepartmanID,  
d.DepartmanID as DepartmaninDepartmanID  
from Calisanlar as c  
inner join Departman as d  
on c.DepartmanID = d.DepartmanID
```

Viewı tekrar çağırılım

View Çağırma

```
select * from vwCalisanlarveDepartmanlari
```

* ifadesi ile sorgu view içerisindeki select ifadesinin sütunlarını çağırıyoruz.

Şimdi view çağırırken * yerine sadece Adi ve DepartmanAdi alanlarını listeyelim

View'da Kolon Filtreleme

```
select Adi,DepartmanAdi from vwCalisanlarveDepartmanlari
```

Hazırlamış olduğumuz viewlar Veritabanı sekmesinin altında views sekmesinde bulunur. Buradan viewlarımızı düzenleyebilir veya silebiliriz

USER DEFINED FUNCTIONS

Sql server içerisinde bize sunulan hazır fonksiyonlar vardır. Aggregate functionlar bunlardan sadece bir kaçıdır. Hazır fonksiyonları kullandığımız gibi kendimizde sql server ortamında kullanabileceğimiz fonksiyonlar tanımlıyabiliyoruz. Fonksiyonların bir diğer kullanım kolaylığı ise gövdelerine dışarıdan parametre alabilmeleridir. User defined funcionları iki bölümde inceliyor olacağız.

1. Scalar-Valued Functions
2. Table-Valued Functions

Dikkat : Parametre bir scope içerisine dışarıdan değer gönderilmesini sağlayan özel değişkenlerdir

SCALAR-VALUED FUNCTIONS

Scalar valued function geriye tek bir hücre dönen fonksiyonlardır. Function tanımlarken function dönüş tipimizi belirliyoruz ve function gövdesinde dönüş tipinde değer çıkarıyoruz.

Ekrana mesaj veren bir fonksiyon yazalım

Fonksiyon Örneği

```
create function fncMesaj()  
returns nvarchar(50) -- function dönüş tipi
```

```
as
begin
    return 'Merhaba Fonksiyonlar' -- function'dan dönen değer
end
```

Fonksiyonumuzu çalıştırdıktan sonra çağırıyoruz.

Fonksiyonu Çağırarak

```
select dbo.fncMesaj()
```

Şimdi çalışanlar tablomuzdaki çalışanların Ad ve Soyad alanlarını birleştiren bir fonksiyon yazalım

Fonksiyon Örneği

```
create function fncBirlestir(@adi nvarchar(50),@soyadi nvarchar(50))
returns nvarchar(100)
as
begin

    declare @tamisim nvarchar(100)
    set @tamisim = Concat(@adi,'',@soyadi)
    return @tamisim

end
```

Daha önceki örneklerimizde alan birleştirme işlemini select ifadelerimizin içerisine yazıyorduk. Artık kendimize bununla alakalı bir modül yazdık ve hem sorgunun okunabilirliği arttırdık hem de ihtiyaç halinde kullanabileceğimiz bir modül oluşturduk. Yazmış olduğumuz fonksiyonu iki ayrı örnekle inceleyelim

Örnek Fonksiyonu Çağırarak

```
select dbo.fncBirlestir('Ekrem','Yıldırım')
```

Sadece select ifadesi ile çağırdığımızda geriye tek bir satır ve sütun olarak döndüğünü görüyoruz. Aynı fonksiyonu şimdi select table ifadesi ile kullanalım

Fonksiyon Kullanımı

```
select
    dbo.fncBirlestir(c.Ad,c.SoyAdi),
    Email,
    TcNo
from Calisanlar as c
```

Ayrıca scalar fonksiyonları sürekli kullandığınız formülleriniz ile kapsülleyip sürekli kullanıma açabilirsiniz.

TABLE-VALUED FUNCTIONS

Table value functions, geriye birden fazla satır ve sütun dönmesi için kullanılır. Tablo dönen fonksiyonları anlamak için viewları tekrar gözden geçirmenizi tavsiye ederim. Tablo dönen fonksiyon yerine view çoğu zaman ihtiyacımızı karşılar. Table valued functionların viewlardan farkı parametre alabilmeleridir. Aşağıdaki iki örneğimizde view ve table valued function tanımlayarak farklılıklarına göz atacağız ve tanımlamalarını göreceğiz. (View tanımlamayı daha önce views bölümünde görmüştük).Örneğimizde CalisanIDsi gönderdiğimiz çalışana erişmek için objelerimizi inceleyeceğiz.

View oluşturma

```
create view vwCalisanlar
as
select * from Calisanlar
```

Oluşturduğumuz view üzerinden CalisanIDsi 1 olan çalışanımızı çağırıyoruz.

View Üzerinden Filtreleme Yapmak

```
select * from vwCalisanlar
where CalisanID = 1
```

Burada oluşturduğumuz view sadece çalışanları listeliyor. View içerisinde CalisanID 1 olan çalışanımızı çağırabiliirdik fakat bu sefer sadece sürekli CalisanIDsi 1 olan çalışan gelecekti. Bu durumda bu view static çalışacak. Bu durumda diğer çalışanlar içinde ayrı ayrı view yazamayacağımız için view dan bütün satırların gelmesini sağladık ve view üzerinden filtreleme yaptık. Şimdi aynı senaryoyu table function ile yazalım

View ile Function Kullanmak

```
create function fncCalisanaGit(@calisanID int)
returns table
as
return (select * from Calisanlar
where CalisanID = @calisanID)
```

Table-Valued functionı incelediğimizde tıpkı scalar functionlar gibi tanımladık. Returns bölümüne table ile dönüş tipinin table olduğunu as den sonra ise return ile select ifademizi tanımladık.

Dikkat : Table-Valued Functions tanımında begin end komutlarını yazmıyoruz

Bu bölümde de user defined functionlar üzerinde durduk. Ayrıca tanımlamış olduğumuz fonksiyonlarımızı veritabanımızın olduğu sekmede Programmability sekmesinin içerisindeki Functions sekmesinin içerisinde buluruz. Burada düzenleme ve silme işlemlerinde gerçekleştirebiliriz.

Stored Procedure

Saklı yordamlar sql serverın tablo objelerinden sonra en önemli objelerinden birisidir. Stored procedure daha önce yazmış olduğumuz komutların kapsüllendiği, server taraflı saklanan bir sql server objesidir. Procedure'ler

functionlar gibi parametre alabilir, tablo döndürebilir, insert,update, delete işlemleri için kullanılabilir sql server job servisi tarafından tetiklenebilir yapılardır. Ayrıca bir procedure sadece tanımlandığı isim ile çağrılarak gövdesindeki komutlar hakkında bilgi vermediği için güvenlik objesi olarakda kullanılır.

Ayrıca stored procedure'ler önemli performans objeleridir. Server tarafında saklandığı için ağ trafiğini azaltır. T-sql yazılan her sorgu parse, optimize, compile, execute aşamalarından geçer. Stored procedure ilk tanımlandığında bu aşamalardan geçer ve sonraki çalıştırma işleminde sadece execute edilir.

Artık örneklerimize başlayalım. Calisanlar tablosuna insert işlemi yapan bir procedure yazalım

Stored Procedure Örneği

```
use SqlOgreniyorum
go
create procedure spCalisanEkle
(
    @ID int,
    @Adi nvarchar(50),
    @SoyAdi nvarchar(50),
    @DepartmanID int,
    @Email nvarchar(11),
    @TcNo nvarchar(11),
    @Maas decimal
)
as
begin
    insert Calisanlar
    (CalisanID,Adi,SoyAdi,DepartmanID,Email,TcNo,Maas)
    values
    (@ID,@Adi,@SoyAdi,@DepartmanID,@Email,@TcNo,@Maas)
end
```

Örneğimizde spCalisanEkle saklı yordamı parametre olarak gövdesine geçiriyor ve gövdede tanımlı insert komutu çalışarak bu parametreler sayesinde gönderilen değerleri Calisanlar tablosuna ekliyor. Aşağıdaki örnekte oluşturduğumuz procedure'u çalıştırıyoruz.

Stored Procedure Kullanmak

```
exec spCalisanEkle 10,'X','Y',1,'x@ba.com','32132131312',1000
```

Yukarıdaki sorgumuzda parametreleri tanım sırasına göre gönderiyoruz. Sorgumuzu çalıştırdığımızda etkilenen satır sayısını sonuç olarak görüyoruz.

Procedure objelerimiz functionlar gibi geriye değer döndürebilirler. Aşağıdaki değer döndüren bir procedure çalıştırıyoruz.

Stored Procedure Örneği

```
create procedure spDepartmanEkle
(
    @ID int,
    @Adi nvarchar(50)
)
as
begin
    declare @kayitVarmi int

    select @kayitVarmi = COUNT(*) from Departman
    where DepartmanID = @ID

    declare @sonuc int
    if @kayitVarmi > 0
    begin
        set @sonuc = 0
    end
    else
    begin
        insert Departman
        (DepartmanID, DepartmanAdi)
        values
        (@ID, @Adi)

        set @sonuc = 1
    end

    return @sonuc
end
```

Procedure incelediğimizde gönderilen @ID parametresi ile departman tablosunda satır sayısını kontrol ediyoruz. Eğer satır sayısı 0 dan büyükse bu ID ile tanımlı bir kayıt olduğu için işlem yaptırmıyoruz. 0 dan büyük değilse else bloğu çalışıyor ve insert işlemini yapıyoruz. If bloğunda @sonuc değişkenine 0 değeri set ediyor, else bloğunda ise 1 set ederek return ile procedure gövdesinden değeri çıkarıyoruz.

Aşağıdaki örneğimizde tanımladığımız procedure çalıştırarak gelen sonucu alalım ve işlem yapıp yapılmadığını kontrol edelim.

Stored Procedure Örneği ve Karar Yapısı

```
declare @kayitYapildimi int
exec @kayitYapildimi = spDepartmanEkle 1, 'Yordam Departman'
if @kayitYapildimi = 0
```

```
begin
    print 'Kayıt işlemi başarılı'
end
else
begin
    print 'Kayıt işlemi başarılı'
end
```

@ID parametresine argüman olarak 1 veriyoruz. Daha önce aynı ID değeri eklediğimiz için procedure'den gelen değer 0 oluyor ve bloğu çalışarak kullanıcıya işlemin başarısız olduğunu gösteriyoruz. Aynı procedure ile başarılı kayıt işlemi yapmak için aşağıdaki komutları çalıştıralım

Stored Procedure ve Karar Yapısı Örneği

```
declare @kayitYapildimi int

exec @kayitYapildimi = spDepartmanEkle 8, 'Yordam Departman'

if @kayitYapildimi = 0
begin
    print 'Kayıt işlemi başarılı'
end
else
begin
    print 'Kayıt işlemi başarılı'
end
```

Result penceresini incelediğimizde Kayıt işlemi başarılı mesajını görüyoruz. Tablomuza select ile çağırdığımızda sonucuda görmüş olacağız.

İşlem Sağlaması

```
select * from Departman
```

Dikkat : Procedure tanımlarken dönüş tipi belirlemiyoruz. Sadece değer döndürecekse return ile değer döndürüyoruz. Dönüş tipi belirlemememizin nedeni procedureler geriye her zaman int değer döner. Eğer procedure içeriden farklı bir tipte değer dönecekse output parameter kullanılır.

Şimdi Calisanlar tablosunda calisanın maas bilgisi güncelleyen bir procedure yazalım.

Stored Procedure Örneği

```
create procedure spMaasGuncelle
(
```

```
@zamTutari decimal,  
@calisanID int,  
@yeniMaas decimal output  
)  
as  
begin  
  
    update Calisanlar  
        set Maas = Maas + @zamTutari  
    where CalisanID = @calisanID  
  
    -- maaşı güncelledikten sonra yeni maaşı @yeniMaas değişkenine atıyoruz  
    select  
        @yeniMaas = Maas  
    from Calisanlar  
    where CalisanID = @calisanID  
end
```

Aşağıdaki örnekte ilk parametreyi gönderiyoruz ve @yenitutar parametresi ile içeriden select ifadesi ile aldığımız değer çıkarıyoruz daha sonra da ekrana yazdırıyoruz

Stored Procedure Kullanımı

```
declare @yenitutar money  
exec spMaasGuncelle 200,1,@yenitutar out  
print @yenitutar
```

TRIGGER

Trigger yine sql içerisinde kullanılan bir server objesidir. Bu zamana kadar yazmış olduğumuz view,function,stored procedure gibi objelerin tetiklenmesi için tarafımızdan çağırılması gerekir. Trigger objeleri ise kendi başına çalışabilen sql server objeleridir. Triggerlar bir olay için yazılır ve o olay gerçekleştiğinde otomatik tetiklenirler. Triggerları ddl trigger ve dml trigger olarak iki bölüme ayırıyoruz. Biz bu bölümde DML trigger üzerinde duracağız.

Dml triggerlar kendi içerisinde 2 bölüme ayrılırlar. Bunlar **after trigger** ve **instead of trigger**lardır. İsimleri ile çalışma mantığı birebir aynıdır. Olaydan sonra tetiklenen trigger ve olay yerine tetiklenen trigger olarak adlandırılırlar.

After Trigger

Insert, update ve delete ifadelerinden sonra tetiklenen triggerlardır. Bu triggerlar bir tablo için yazılır ve tabloya bu işlemten birisi gerçekleştiğinde otomatik tetiklenirler. Öncelikle Calisanlar tablomuzu email alanı null olabilir şekilde güncelleyelim

Trigger Kullanımı için Tablo Güncellemesi

```
alter table Calisanlar
```



```
alter column Email nvarchar(11) null
```

Tablomuzu güncelledikten sonra bu tabloya bir insert işlemi gerçekleştiğinde çalışan adı ile bir email değeri oluşturup email alanına otomatik eklenmesini sağlayalım.

Trigger Örneği

```
use SqlOgreniyorum
go
create trigger trgMailEkle
on Calisanlar
after insert
as
begin
    declare @adi nvarchar(11)
    declare @calisanID int

    select @adi = Adi, @calisanID = CalisanID from inserted

    update Calisanlar
    set Email = CONCAT(Substring(@adi,1,2),'@ba.com')
    where CalisanID = @calisanID
end
```

Dikkat : inserted tablosu trigger içerisinde erişilebilen ve hangi tablo için trigger yazıldıysa o tablo için eklenen son satırı depolayan özel bir tablodur.

trgMailEkle adlı trigger Calisanlar tablosuna insert işlemi gerçekleştiğinde tetiklenecek ve inserted tablosundan insert edilen Adi ve Idsini alarak ID'ye göre Çalışanın Email alanını isim@ba.com olarak güncelleyecek. Şimdi gidip bu tablo için bir insert komutu yazıyoruz.

Ekleme İşlemi

```
insert Calisanlar
(CalisanID,Adi,SoyAdi,TcNo,Maas,DepartmanID)
values
(11,'Nancy','Nancy','56432198704',750,1)
```

Yukarıdaki insert komutunu çalıştırdığımızda bizim tarafımızdan bir insert işlemi, trigger tarafından ise bir update işlemi gerçekleşti.

Şimdi Calisanlar tablosundan select ile Nancy olan çalışınımızı isteyelim ve Email alanı kontrol edelim.

Tablo Listeleme

```
select * from Calisanlar
where Adi ='Nancy'
```

Gördüğünüz gibi Adi Nancy olan çalışmamızın Email alanı trigger tarafından güncellenmiş.

Instead Of Trigger

Instead of Trigger insert, update, delete komutlarının yerine çalışan trigger türüdür.

Aşağıdaki örnekte departman tablosundan kayıt silinmesini engelleyen bir trigger yazıyoruz.

Instead Of Kullanımı

```
use SqlOgreniyorum
go
create trigger trgKayitSildirme
on Departman
instead of delete -- delete yerine çalışacak
as
begin
    print 'Bu tablodan kayıt silemezsin'
end
```

Örneğimizde Departman tablosundan delete komutu ile kayıt silineceği zaman trigger gövdesindeki komutlar delete komutu yerine çalışacaktır.

TRANSACTION

Transaction sql sorgularımızı takip eden bir iş birimidir. Transaction ile önemli sorgularımızı özellikle dml (select,update,insert,delete) işlemlerimizin takipe alınarak işlem sonucuna kadar izleriz. Transaction ile başlatılan sorgular transactionı onaylamadan tamamlanmaz. Komut izlemeyi **begin transaction** ile başlatır, **commit transaction** ile onaylar **rollback transaction** ile başa döneriz.

Aşağıdaki örnekte transaction ile bir insert işlemi yapıyoruz.

Transaction Kullanımı

```
begin transaction
insert Departman
(DepartmanID,DepartmanAdi)
values
(104,'DB')
```

Sorguyu çalıştırdığımızda Departman tablosuna yukarıdaki kayıtları ekliyoruz ve kontrol için tekrar bir select sorgusu yazıyoruz.

Veri Listeleme

```
select * from Departman
```

Sonuca baktığımızda DB isimli departmanın eklendiğini görüyoruz. Transaction tanımında begin transaction ile başlatılan işlemin onaylamadan tamamlanmadığını söylemiştik fakat sonuç kümesine baktığımızda verinin tabloya eklendiğini görüyoruz. Şimdi daha iyi anlamak adına tekrar begin transaction yapmadan bir insert işlemi daha yapalım

Insert İşlemi

```
insert Departman  
(DepartmanID,DepartmanAdi)  
values  
(105,'YZLM')
```

Listeleme İşlemi

```
select * from Departman
```

İlk olarak insert sorgusunu daha sonrada select sorgularını çalıştırdık. Tablomuza tekrar bir YZLM departmanının eklendiğini görüyoruz. Şimdi aşağıdaki sorguyu çalıştıralım.

rollback transaction

```
select * from Departman
```

Sonucu incelediğimizde rollback transaction bölümüne kadar eklenen verilerin kaybolduğunu gördük. Bu da demek oluyor ki transaction ile yapılan işlemin, işlem başarılıysa commit edilmesi gerekir. Şimdi tekrar transaciton ile bir insert işlemi yapalım ve commit edelim.

Commit Transaction

```
begin transaction  
insert Departman  
(DepartmanID,DepartmanAdi)  
values  
(105,'YZLM')  
commit transaction
```

Artık bu işlemten sonra rollback transaction çalıştırsak bile işlemler geri alınmaz ve bir transaction başlatılmadığı için hata alırız.

SQL SERVER 2016 YENİLİKLERİ

DYNAMIC DATA MASKİNG

Sql Server'ın güvenlik yeniliklerinden birisidir. Tablodaki bazı sütunları kullanıcı rollerine göre önemli olduğu için göstermek istemeyebiliriz. Burada şimdiye kadar bu tür senaryoları view,function,procedure gibi objeler ile yapıyorduk. Dynamic Data Masking ile kullanıcı direk tablo üzerinden sorgu yapabilir fakat belirttiğimiz kolonları

maskeli olarak görmesini sağlar. Maskeleye yeniliğinin özelliği gerçek veriyi bozmaksızın sadece select yetkisi verilen kullanıcıya maskeleyerek gösterir. Yeni gelen maskeleye desteğiyle bize 3 adet fonksiyon sunuluyor. Bunlar **default**, **email** ve **partial** fonksiyonlarıdır. **Default** maskeleye metinsel, sayısal ve tarih alanlarında kullanılabilir. Metinsel alanlarda xxx sayısal alanlarda 0 tarihsel alanlarda ise tarih alanlarının default değeri gösterilir. **Email** fonksiyonu ise veriyi email formatında gösterir. Bu iki fonksiyon hazır fonksiyonlardır. Partial fonksiyonu ise maskeleye yöntemini bize bırakır.

Bununla alakalı bir kaç örnek yapalım.

İlk olarak bir tane loginsiz kullanıcı oluşturuyoruz.

```
create user ikCalisani without login
```

Daha sonra bu kullanıcıya Calisanlar tablosu için select yetkisi veriyoruz.

```
grant select on Calisanlar to ikCalisani
```

Kullanıcı işlemlerimizi hallettikten sonra ikCalisani kullanıcının Calisanlar tablosundaki maaş alanını görmesini engellemek için Calisanlar tablosunun maaş alanını default masking yöntemiyle maskeliyoruz.

```
alter table Calisanlar  
  alter column maas add masked with (function='Default()')
```

Daha sonra mevcut kullanıcımız ile select ifadesiyle tablomuzu listeliyoruz

```
select * from Calisanlar
```

Sonucu incelediğimizde maaş alanı maskesiz olarak karşımıza çıkıyor. Çünkü biz select yetkisini sadece ikCalisani kullanıcıya vermiştik. Şimdi ikCalisani login olmuş gibi sorgumuzu çalıştırıyoruz ve sonucu listeliyoruz.

```
execute as user = 'ikCalisani'  
  select * from Calisanlar  
revert;
```

Result penceresini incelediğimizde maas alanının ikCalisanina 0.00 olarak gösterildiğini görüyoruz.

Şimdi tekrar ikCalisanina TcNo alanını maskeleyerek gösterelim.

```
alter table Calisanlar  
  alter column TcNo add masked with (function='Default()')
```

Tekrar ikCalisani ile select yapalım

```
execute as user = 'ikCalisani'  
  select * from Calisanlar  
revert;
```

Sonucu incelediğimizde TcNo alanlarının xxxx olarak maskelendiğini görüyoruz. Şimdi Email alanını email fonksiyonu ile maskeleyelim.

```
alter table Calisanlar
    alter column Email add masked with (function = 'Email()')
```

Ardından select ifademizi ikCalisani ile çalıştırıyoruz

```
execute as user = 'ikCalisani'
select * from Calisanlar
revert;
```

Email alanını da maskeli bir şekilde görmüş olduk. Dynamic Data Masking öneli bir güvenlik yeniliğidir. Örnekleri incelediğimizde kısa tanımlamalar ile view gibi objeler oluşturmada veri güvenliğini sağlamış olduk.

NATIVE JSON

Native json özelliğine değinmeden önce Json nedir kısaca açıklayalım. Json bir data transfer aracı ve dilidir. Özellikle platformlar arası veri transferinde bağımsız dil desteği ile xml diline bir rakip olarak düşünülebilir. Özellikle web ve mobil uygulamalarda veri transfer aracı olarak karşımıza çıkmaktadır. Bunun nedeni web teknolojilerinin gelişmesiyle farklı dillerde yazılan uygulamalar çok haberleşme ihtiyacı duyuyor. Bu noktada json düşük maliyet ve güçlü veritipi sayesinde bu haberleşme sürecinde ara bir dil görevi görüyor. Sql Server 2016 ile yerel olarak(SQL SERVER ortamında) json data özelliğini desteklemeye başladı. Özellikle web api, servis gibi soa mimarilerinde elimizdeki veriyi class veya list yerine json'a çevirerek gönderme ihtiyacı duyuyorduk. Artık 2016 ile birlikte veriyi direk sql serverdan json olarak çekebiliyoruz.

Bununla alaklı örneklerimizi aşağıda inceleyelim.

Departman tablosundaki verilerimi listeleyelim

```
select * from Departman
```

Json formatta veriyi listeleme

```
select * from Departman
FOR JSON AUTO;
```

Sonucu incelediğimizde verilerin satır sütun olarak değilde { } süskü parantezler içerisinde Sütun:Değer olarak geldiğini görüyoruz. Burada { } süslü scope'ların herbiri bir satır olarak yorumlanır. Bunu json olarak yorumlarsakta bunun bir json objesidir. [] köşeli parantezler ise birden fazla nesnenin bir arada tutulduğunu ifade eder. Bunuda c# Introductiondaki diziler olarak düşünelim.

Json data kullanıcı tarafı yorumlaması zor bir formattır. Şimdi mevcut bir json datayı tablo olarak yorumlayalım.

```
SELECT * FROM OPENJSON('{ "DersAdi": "Sql Server", "Konu": "Json" }')
```

Native json yine önemli bir Sql Server yeniliğidir. Bu bölümün faydalarını ilerideki konularımızda daha detaylı konuşuyor ve anlıyor olacağız.

ROW LEVEL SECURITY

Row level security (Satır Seviyesi Güvenlik) özelliği tablodaki veriyi sadece ilgili kullanıcıya gösterme yeniliğidir. Tablo oluştururken select yetkisi verdiğiniz kullanıcıların ilgili verileri görmesini istiyorsanız iki adet fonksiyon ile bu özelliği aktif olarak kullanabiliyorsunuz.

Row Level Security'i anlamak için örnek üzerinden gidiyor olacağız.

Öncelikle "SqlOgreniyorum" veritabanına yeni bir tablo oluşturalım.

Tablomuz çalışanların çalışmış oldukları projelerini tutan ve hak edişlerini hesaplayan bir tablo olsun

```
create table Proje
(
    ID int identity(1,1),
    ProjeAdi nvarchar(50) not null,
    ProjeBaslangicTarihi datetime default getdate(),
    ProjeBitisTarihi datetime,
    HakEdis money,
    Calisan nvarchar(50)
)
```

Daha sonra iki adet sql kullanıcısı oluşturalım ve oluşturduğumuz proje tablosuna select yetkisi verelim.

Rıdvan ve Ekrem isimli iki sql kullanıcısı oluşturuyoruz

```
create user ridvan without login
create user ekrem without login

grant select on Proje to ridvan
grant select on Proje to ekrem
```

Kullanıcılarımızı oluşturduktan sonra bir procedure tanımlayalım. Veritabanı yöneticimiz bizim verdiğimiz bir procedure ile projelerinin adı, başlangıç, bitiş tarihlerini ve çalışan bilgisini sql server ortamında girecektir. Daha bir trigger yardımıyla hak ediş hesaplaması yapacağız .

Öncelikle projeler tablosuna yöneticimizin insert yapabilmesi için bir procedure yazıyoruz.

```
create procedure spProjeEkle
(
    @ProjeAdi nvarchar(50),
    @BaslangicTarihi date,
    @BitisTarihi date,
    @Calisan nvarchar(50)
)
as
begin
```

```
insert Proje
(ProjeAdi,ProjeBaslangicTarihi,ProjeBitisTarihi,Calisan)
values
(@ProjeAdi,@BaslangicTarihi,@BitisTarihi,@Calisan)
```

```
end
```

Şimdi yöneticimizin insert yaptığında çalışanların hakedişlerini hesaplayabilmek için insert işleminden sonra çalışan bir after trigger yazıyoruz.

```
create trigger trgHakEdisHesapla on Proje
after insert
as
begin
    declare @gunSayisi int
    declare @ID int
    -- Dateiff fonksiyonu iki tarih arasındaki farkı bulur
    select @ID = ID, @gunSayisi = DATEDIFF(day,ProjeBaslangicTarihi,ProjeBitisTarihi) from inserted

    update Proje
        set HakEdis = @gunSayisi * 100
    where ID =@ID

end
```

Şimdi artık sırasıyla bu kullanıcılarımız için çalışmış oldukları projeleri ekleyelim.

```
exec spProjeEkle 'x şirketi yazılım danışmanlığı','01/01/2016','01/10/2016','Rıdvan'

exec spProjeEkle 'xy şirketi yazılım danışmanlığı ve eğitim','01/11/2016','01/15/2016','Rıdvan'

exec spProjeEkle 'x şirketi yazılım danışmanlığı','01/05/2016','01/06/2016','Ekrem'

exec spProjeEkle 'xy şirketi yazılım danışmanlığı ve eğitim','01/07/2016','01/11/2016','Ekrem'
```

Yukarıdak örneğimizde daha önce yazmış olduğumuz store procedure ile bilgileri proje tablosuna yazıyoruz ve alttaki select ifadesi ile sorgumuzu çalıştırıyoruz

```
select * from Proje
```

Sonucu incelediğimizde bütün çalışanlarımızın proje bilgilerine ulaştık. Şimdi daha önce oluşturduğumuz kullanıcılar ile select yapalım.

```
execute as user ='rıdvan'  
select * from Proje  
revert;  
  
go  
  
execute as user ='ekrem'  
select * from Proje  
revert;
```

Sonucu incelediğimizde rıdvan ve ekrem kullanıcılarımızın bütün verilere eriştiğini görüyoruz. Buraya dynamic data masking özelliği yerine row level security kullanarak kullanıcının sadece kendisiyle ilgili yani çalışmış olduğu projeler erişimini sağlayacağız. Öncelikle RLS yapabilmemiz için bir adet security policy eklemiz gerekiyor. Ancak burada filtreleme işlemi yapılacağı için bizden table valued function isteyecek. Öncelikle fonksiyonumuzu yazalım.

```
create function CalisanKontrol  
(  
    @kullanici nvarchar(50)  
)  
returns table  
with schemabinding  
as  
    return select 1 as UserRowLevel  
    where @kullanici = USER_NAME()
```

CalisanKontrol fonksiyonu bizim çağıracağımız bir fonksiyon değil. Bu fonksiyon oluşturulan security nesnesi tarafından çağrılacaktır. Çalışma mantığı olarakta Gönderilen Parametre ile mevcut kullanıcı adını kontrol edecek. Eğer karşılaştırılan değerler birbirine eşitse 1 yani true değilse null dönecek. Şimdi aşağıdaki sorgumuzla az önce ifade ettiğimiz security policy nesnemizi oluşturalım.

```
CREATE SECURITY POLICY RowFilterPolicy  
ADD FILTER PREDICATE dbo.CalisanKontrol(Calisan)  
ON dbo.Proje  
WITH (STATE = ON)
```

Oluşturulan nesne daha önce yazdığımız fonksiyonumuza Calisan sütunundaki değeri parametre olarak dönecek ve fonksiyondan gelen geri dönüş tipine göre veriyi gösterecek. Son olarak bu işlemleri yaptıktan sonra aşağıdaki komutlarımız ile tekrar sorgumuzu çalıştıralım. Sonuca baktığımızda sorguyu çalıştırdığımız kullanıcı için sadece kendisiyle alakalı verilerin geldiğini göreceğiz.

```
execute as user ='rıdvan'  
select * from Proje
```



```
revert;  
  
go  
  
execute as user = 'ekrem'  
select * from Proje  
revert;
```

TEMPORAL TABLES

Temporal Table tablolardaki update, delete işlemlerini izlemek için kullanılan yeni bir özelliktir. Bir temporal table oluşturduğumuzda iki adet tablo tanımlanır. Temporal table tanımladığımızda iki adet tablo oluşur. Bunlar **SYSTEM-VERSIONED TABLE** ve **HISTORY TABLE** 'dır. System versioned table gerçek verinin tutulduğu tablodur. Bu tablo üzerinde yapılan bir update veya delete işlemi sql server tarafından history table'a taşınır. İkisi arasındaki değişiklikler bu tablolar üzerinden karşılaştırılır. Daha önceki sql server sürümlerinde bu işlemler manuel tarafımızdan yapılırdı. Özellikle update ve delete işlemleri geri alınamadığından (daha doğrusu geri dönüşü kolay olmadığından) bu tablolar sql server kullanıcıları için önemli bir özelliktir.

Temporal table oluşturabilmek için, oluşturulacak tabloda iki adet datetime2 tipinde sütuna ihtiyacımız vardır. Bu alanlar verinin değişim sürecini izlemek için kullanılır.

Örnek olarak bir tane ürün tablosu oluşturalım. Bu tabloda stok hareketleri sürekli değişsin ve biz bu değişimleri history table üzerinden inceleyelim.

```
create table OrnekTablo  
(  
    ID int not null identity(1,1) primary key,  
    Adi nvarchar(50),  
    BaslangicTarihi datetime2 generated always as row start not null, -- değerler sql server tarafından eklenecek  
    BitisTarihi datetime2 generated always as row end not null, -- değerler sql server tarafında eklenecek  
  
    Period for system_Time(  
        BaslangicTarihi, BitisTarihi -- verinin geçerlilik süresi hangi alanlar üzerinden yapılacak  
    )  
)  
with(system_Versioning = on (History_Table = dbo.OrnekHistory))
```

Bu işlemden sonra sqlOgreniyorum veritabanımızın Tables sekmesini açtığımızda OrnekTablo sekmesinin içerisinde OrnekHistory table olarak tablo tanımlandığını göreceksiniz.

Aşağıdaki örnekte oluşturduğumuz tabloya birkaç insert işlemi yapalım

```
insert OrnekTablo  
(Adi)  
values  
( 'A' ),  
( 'B' ),  
( 'C' )
```

Şimdi OrnekTablo ve History tablosu için select yapıyoruz

```
select * from OrnekTablo
select * from OrnekHistory
```

Sonucu incelediğimizde OrnekTabloda 3 satır varken History tablosunda henüz veri olmadığını gördük. History tablosu Ornektablosunda değişiklik yapıldığında eski verilerin taşınacağı tablodur. Henüz değişiklik yapmadığımız için tabloda kayıt olmaması normaldir. Aşağıdaki kodu çalıştırıp ID si 1 olan satırı güncelleyelim.

```
update OrnekTablo
set Adi = 'ABC'
where ID = 1
```

Güncelleme işlemini yaptıktan sonra tekrar OrnekTablo ve History tablosu için select yapıyoruz. Daha sonra result penceresini inceleyelim. Sonuca baktığımızda IDsi 1 olan A verisi ABC olarak değişmiş ve sql server tarafından history tablosuna A değeri satır olarak taşınmış. Başlangic ve bitisTarihi alanlarını incelediğimizde verinin hangi tarihler arasında geçerli olduğunu görüyoruz.

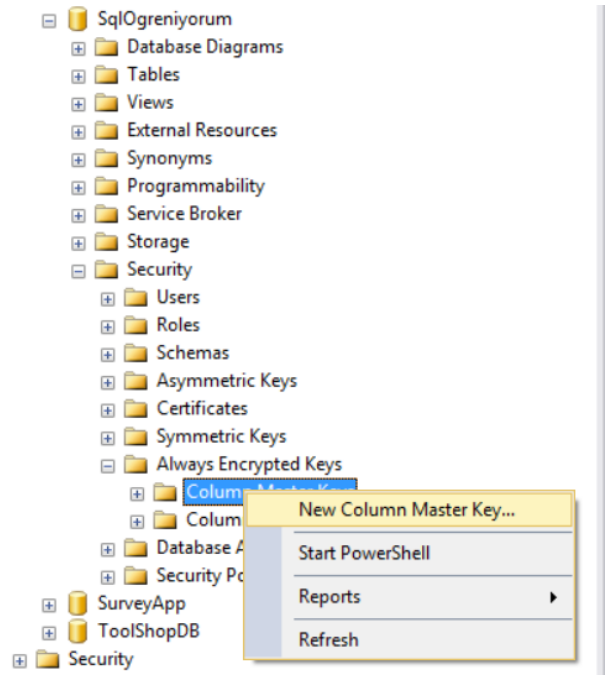
```
select * from OrnekTablo
select * from OrnekHistory
```

ALWAYS ENCRYPTED

Always Encrypted özelliği önemli verilerin güvenliğini sağlamak için oluşturulan yeni bir güvenlik yöntemidir. Bu özelliğin en önemli kısmı veriyi herkese karşı şifrelemesidir. Yani veritabanında veri güvenliği için DBA dahil herkese karşı şifreler.

Always encrypted özelliğini aktif hale getirebilmemiz için Veritabanımıza sağ tık yaparak, security sekmesinden Always Encrypted Keys sekmesi altından yeni bir Column Master Keys'e sağ tık yaparak New Column Master Key seçeneğini tıklıyoruz. Daha sonra gelen pencereden name bölümüne bir key ismi veriyoruz. İsim verdikten sonra Bu pencerede name altında olan Key Store select listesi içerisinde Local Machine seçeneğini seçiyoruz. Bu şifrenin nerede saklanacağı ile ilgilidir..

Bu işlemi şifreleme yapacağımız alanları şifrelemek için yaptık. Yani şifreleme yapılacak sütunlar şifrelenmiş bir master key'e ihtiyaç duyar.

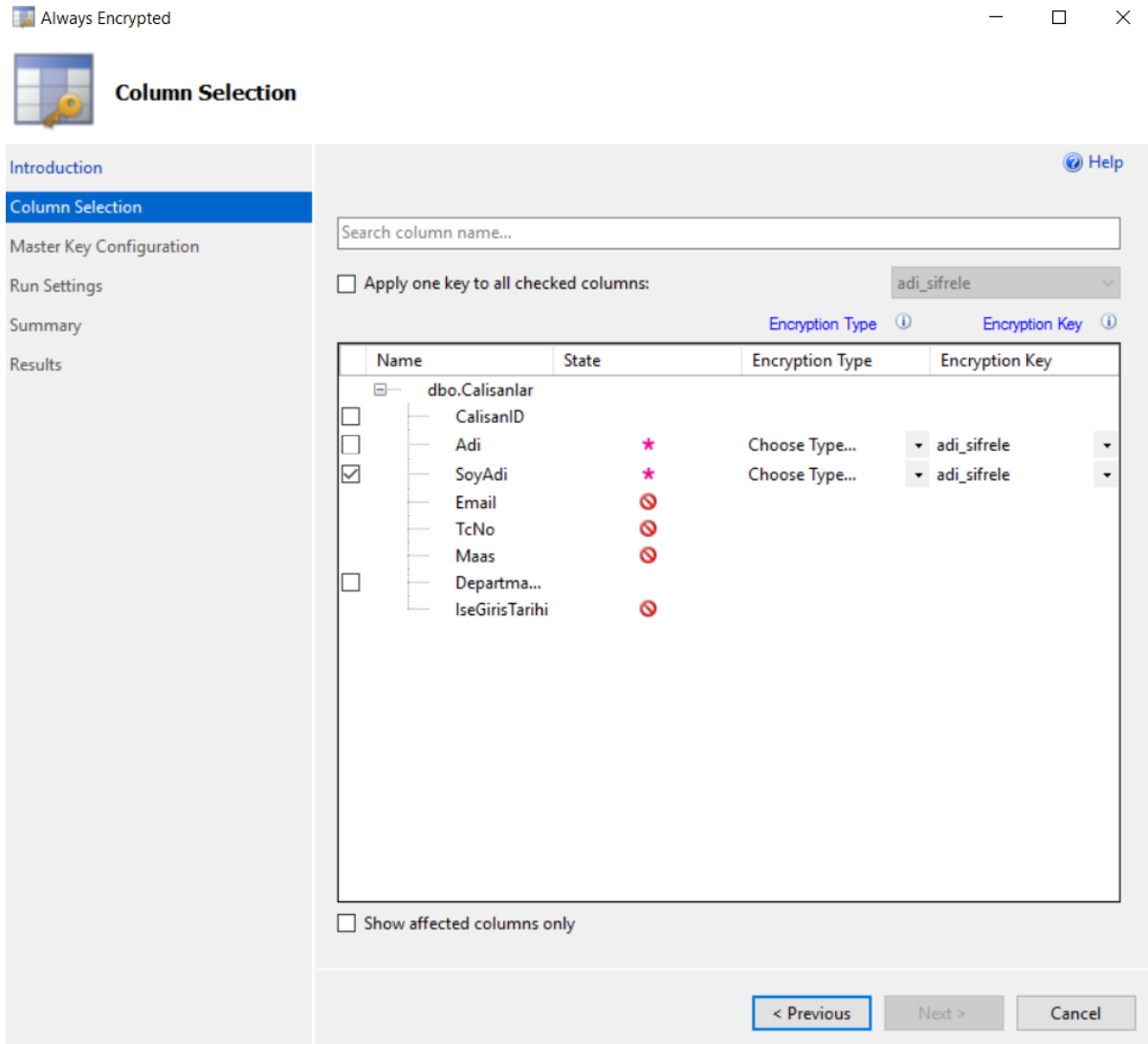


Daha sonra Column Encryption Keys' e sağ tık yaparak New Column Encryption Key diyerek gelen pencereden Key adı verelim ve bu şifre alanını hangi master key şifreleyecek belirlemek için name altındaki Column Master

Keyi seçelim ve Ok diyerek penceremizi kapatalım. Bu işlemlerden sonra artık column encrypted seçeneğini aktif hale getirelim. Yani hangi tablo üzerinde hangi sütun şifrlenecek bunu belirleyeceğiz.

Veritabanı sekmemizi açarak Calisanlar tablomuza sağ click yapıp Encrypt Columns'ı seçelim.

Daha sonra karşımıza bir şifreleme sihirbazı gelecek. Bu pencerede introduction bölümün next ile geçelim ve column section bölümüne geçelim. Aşağıdaki pencerede şifreleme yapacağımız sütunlar bulunuyor. Önce şifrelenecek sütun veya sütun isimlerini seçiyoruz.



Daha sonra Encryption Type bölümünden Şifreleme türünü seçmemiz gerekiyor. İki tür şifreleme algoritması mevcuttur. Bunlar;

Deterministic : Veriyi sürekli aynı yöntemle şifreler.

Randomized : Sürekli farklı şifre üretir.

Daha sonra Encryption Key alanından adi_sifreli Keyi seçiyoruz. Daha sonra next diyoruz Results penceresine geldiğimizde işlemin tamamlanması birkaç saniye sürebilir Cancel diyerek penceremizi kapatıyoruz.

Daha sonra query penceresi açıp sorgumuzu çalıştırdığımızda aşağıdaki gibi soyadi alanını şifrelenmiş görüyoruz.

```
4 Use SqlOgreniyorum
5 select * from Calisanlar
```

158 %

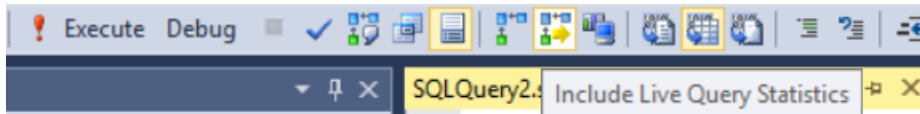
Results Messages

	CalisanID	Adi	SoyAdi	Email	TcNo	Maas	DepartmanID	IseGirisTarihi
1	1	Ekrem	0x01153289FC1DE9AE2134E3DF7F507B21E6CEF5895A408D9...	e@ba.com	12345567898	1500,00	2	2017-02-26
2	2	Alp	0x0137707C3A6EFA906ACC111C8CCCEC521B05872612D4392...	a@ba.com	23345567898	3500,00	1	2017-02-26
3	3	Rıdvan	0x0175369B74F40FAD10BB5C3292F316DD153AF3BB465BDEE...	r@ba.com	33345567898	500,00	2	2017-02-26
4	4	Mert	0x016B04C9118F934971ABEB24D52655353FB2C0DD8B148CF...	m@ba.com	43345567898	1500,00	2	2017-02-26
5	5	Sabri	0x0152A3B51E7060EE1970A00A8A51E6153484BCF33F713039...	s@gs.com	98765432191	500,00	3	2017-02-26
6	10	X	0x01393F944D7573A5730A91EE721E2EEF8FAA3303711F043...	x@ba.com	32132131312	1000,00	1	2017-02-27
7	11	Nancy	0x01C4EF53C3982DDDD6F3D29CD498C62A0F4116020E5CE2...	Na@ba.com	56432198704	750,00	1	2017-02-27

LIVE QUERY STATISTICS

Live query statistics özelliği sayesinde sorgumuzun canlı çalışma planını izleyebiliriz. Daha önce Actual Execution Plan ile bu özelliği kullanıyorduk. Fakat bu seçenek sorgu bittikten sonra aktif oluyordu. Live query statistics sayesinde yazılan sorgunun çalışma zamanı planını izleyerek sorgunun ne kadarlık bölümünün gerçekleştiğini görebiliyoruz. Bu seçeneği aktif edebilmek için tool bar'dan Include Live Query Statistics düğmesine tıklayarak aktif hale getirip sorgumuzu çalıştırdığımızda query penceresimizin altında Live Query Statistics sekmesi gelecek ve sorgunun çalışma süresini ne kadarlık bölümünün tamamlandığını göreceğiz.

Include live Query Statistics seçeneğini aktif hale getirelim



Daha sonra Calisanlar tablomuzu select ile çağıralım. Query penceresimizin altındaki Live Query Statistics penceresini incelediğimizde sorgumuzun Index Scan yaptığını görüyoruz. Bu pencerenin özelliği sorgu çalışırken bize planı canlı olarak göstremesidir. Sorgunun tamamlanan bölümleri düz çizgi devam eden bölümleri ise şerit halinde gösterilir

```
select * from Departman
```

