

veri tabanı

SELECT	Veri sorgulama
INSERT	Veri ekleme
UPDATE	Veri güncelleme
DELETE	Veri silme
CREATE	Nesne (tablo, veritabanı vs.) oluşturma
ALTER	Nesne değiştirme
DROP	Nesne silme
WHERE	Şart belirtme
ORDER BY	Sıralama
GROUP BY	Gruplama

GROUP BY	Gruplama
HAVING	Gruplara filtre
JOIN	Tablo birleştirme
UNION	Sorgu birleştirme
DISTINCT	Tekil kayıtlar
TOP	İlk N kaydı getirme
LIKE , IN , BETWEEN , IS NULL	Koşul ifadeleri
PRIMARY KEY , FOREIGN KEY	Anahtarlar
VIEW , INDEX , STORED PROCEDURE , TRIGGER	Gelişmiş nesneler

SPACE, ISNULL, COALESCE, CAST, NULLIF, MIN, MAX ve STRING ve DATE Fonksiyonları.

Değişken tanımlama IF, ELSE, CASE, WHILE yapıları ve örnekleri.

USE ve GO komutları.

Index kavramı, index çalışma yapısı, yönetilmesi.

Index öncesi ve sonrası tablo sorgu hız kıyaslaması.

Unique Index, Clustered Index, Filtered Index, Full Text.

Index Rebuild.

Fonksiyon oluşturma, kullanımı ve örnekleri.

User-Defined Functions ve Built-In Function ve örnekleri

View oluşturma, kullanımı ve örnekleri.

Geçici tablo kullanımı

INT	Tamsayı (4 byte)
BIGINT	Çok büyük tamsayı
SMALLINT	Küçük tamsayı
TINYINT	0-255 arası
DECIMAL(5,2) / NUMERIC	Ondalıkli sayı (5 basamak, 2'si virgülden sonra)
FLOAT	Yaklaşık ondalık
BIT	0 ya da 1 (true/false)

tanımlama yaparken "declare" komutu kullanılıyor.

```
DECLARE @yas INT = 25;
DECLARE @fiyat DECIMAL(5,2) = 45.90;

DECLARE @yas INT = 21;
SELECT @yas AS Yas;
-- burada yapılan şey @yas adında bir değişken tanımlayıp
--select ile ekrana bastırmak ve "Yas" adı altında veri gösteriyorum.
--@yas değişkenine geçici olarak "Yas" adını ver.
        --Yas
        --21
```

CHAR(n)	Sabit uzunlukta karakter
VARCHAR(n)	Değişken uzunlukta karakter
TEXT	Çok uzun metin (deprecated - eski)

```
DECLARE @ad VARCHAR(50) = 'Ahmet';
en fazla 50 karakterlik alan ayır demek
ama bunun değişkenliğinde sağlıyor sabit değil yani
eğer sabit uzunluk isteniyorsa char(n) tanımı kullanılabilir.
```

DATE	Yıl-Ay-Gün
DATETIME	Tarih + saat
TIME	Sadece saat
DATETIME2	Daha hassas datetime

```
DECLARE @dogumTarihi DATE = '2003-10-10';
```

AS , SQL'de bir **alias (takma ad)** vermek için kullanılır. Bu takma ad:

- Kolonlara,
- Değişkenlere,
- Fonksiyon sonuçlarına,
- Tablo adlarına bile verilebilir.

```
SELECT sütun_adı AS YeniAd FROM tablo_adi;
```

```
SELECT GETDATE() AS TamTarihSaat,
```

```
CONVERT(DATE, GETDATE()) AS SadeceTarih;
```

--GETDATE() → sistemden tarih + saat alıyor → buna "TamTarihSaat" adını veriyoruz.

--CONVERT(DATE, GETDATE()) → sadece tarihi alıyor → buna "SadeceTarih" adını veriyoruz.

--Bu sadece çıktıda isim görünsün diye yapılan bir şey, verinin kendisini değiştirmez.

FONKSİYONLAR;

SPACE(n): belirtilen kadar boşluk dönmesini sağlıyor.

ISNULL(expr, replacement): null yerine verilen değeri koyar?

COALESCE(expr1,expr2,...): ilk null olmayan değeri döndürür.

CAST VE CONVERT: veri tipini dönüştürür.

NULLIF(a,b): iki değer eşitse null döndürür, değilse birinci değeri dönmesini sağlar.

MİN,MAX: en küçük, en büyük değerın dönmesini sağlar.

```
SELECT 'Ad:' + SPACE(5) + 'Ahmet';
```

```
-- Sonuç: 'Ad:   Ahmet' (5 karakter boşluk bıraktı)
```

```

SELECT ISNULL(Telefon, 'Telefon girilmedi') AS TelefonDurumu
FROM Ogrenciler;
-- telefon kısmı null ise telefon girilmedi diye not düşüyor.

SELECT COALESCE(NULL, NULL, 'Ali', 'Ahmet'); -- 'Ali'

SELECT CAST(123 AS VARCHAR(10)); -- '123'
SELECT CONVERT(DATE, GETDATE());
-- Yalnızca tarihi döndürür date yerine getdate kullanımını sağladı.
--getdate() şuanki tarih ve saat bilgisi döndürür.
--convert(date,getdate()) dediğimde sadece tarih kısmını alır.

SELECT NULLIF(5, 5); -- NULL
SELECT NULLIF(5, 3); -- 5

SELECT MIN(Maas), MAX(Maas) FROM Calisanlar; -- min maas ve max maası çalışanlar tablosu

```

STRING FONKSİYONLARI:

len(s): uzunluk veriyor.

upper(s). büyük harfe dönüşüm

lower(s): küçük harfe dönüşüm sağlar.

left(s,n): string ifadenin sol tarafından n tane karakter alınması

right(s,n): string ifadenin sağ tarafından n tane karakter alınması

substring(s,start,length): parça alınması

replace(s,old,new): değiştirmek için

```

SELECT
LEN('Ahmet') AS Uzunluk,
UPPER('ahmet') AS Buyuk,
LOWER('AHMET') AS Kucuk,
LEFT('Ahmet', 2) AS Sol2, -- ah
RIGHT('Ahmet', 2) AS Sag2, --et
SUBSTRING('Ahmet', 2, 3) AS Orta3,
-- sqlde indisler 1den başlıyor. 1.indis A oluyor
--hme 2.indisten başlayarak 3 tane char alıyor.

REPLACE('Merhaba Ali', 'Ali', 'Veli') AS YeniMetin;
--verilen string içinde (merhaba ali) eski stringi
--buluyor(ali) ve yenisiyle(veli) değiştiriyor

```

DATE FONKSİYONLARI

getdate(): şuanki tarih ve saat döndürür

getutcdate(): utc zamanı

dateadd(part,n,date): tarihe ekleme yapar

datediff(part,start,end): tarih farkı

format(date,format): şekilli tarih döndürür

!!! **FORMAT** ile **CONVERT** farkı sınavda çıkar: biri görünüm, biri tip dönüşüm.

!!! **DATENAME** yazı döndürür → dikkat!

```
SELECT GETDATE() AS Suan; --2025-04-07 16:23:45.320
SELECT GETUTCDATE() AS UTCZaman;--2025-04-07 13:23:45.320 -- (3 saat fark olabilir)
SELECT GETDATE() AS Bugun,
DATEADD(DAY, 7, GETDATE()) AS YediGunSonra;
/*Bugun      → 2025-04-07 16:23:45.320
YediGunSonra → 2025-04-14 16:23:45.320
*/
SELECT DATEDIFF(DAY, '2025-01-01', GETDATE()) AS GunFarki,
DATEDIFF(YEAR, '2003-10-10', GETDATE()) AS Yas;
/*GunFarki → 97
Yas → 21*/
SELECT FORMAT(GETDATE(), 'dd/MM/yyyy') AS GunAyYil,
FORMAT(GETDATE(), 'dddd, MMMM yyyy') AS TamYazi;
/*GunAyYil → 07/04/2025
TamYazi → Pazartesi, Nisan 2025 */
```

DEĞİŞKEN TANIMLAMA, IF, ELSE, CASE, WHILE

değişken tanımlama;

```
DECLARE @degiskenAdi VeriTipi;
SET @degiskenAdi = Değer;

DECLARE @sayi INT;
SET @sayi = 10;
SELECT @sayi AS Deger;
-- sayı adında int veritipinde bir değişken tanımlanır
--sayı değişkenime 10 değeri atanır. ve select ile gösterilir.
```

```
DECLARE @mesaj VARCHAR(50) = 'Merhaba!';
SELECT @mesaj AS Cikti;
--aynı şekilde mesaj adında bir değişkenim var ve 50 karaktere sahip
--ve değeri "merhaba" select ile gösterirken çıktı adı altında mesajımı gösteriyorum..
```

IF ELSE;

```
IF (koşul)
BEGIN
-- kod bloğu 1
END
```

```
ELSE
BEGIN
-- kod bloğu 2
END
```

```
DECLARE @yas INT = 20;
```

```
IF @yas >= 18
BEGIN
PRINT 'Reşit';
END
```

```
ELSE
BEGIN
PRINT 'Çocuk';
END
```

/*yas adında bir değişken tanımladım ve 20 değerini atadım.
koşul olarak da eğer 18den büyükse reşit
olduğunu değilse çocuk olduğunu söylüyorum.
!!! burda dikkat edilmesi gereken şeyler BEGIN ve END satırları */

--eğer tek satır olacaksa begin ve end kullanımına gerek yok.

```
DECLARE @x INT = 10;
DECLARE @y INT = 5;
```

```
IF @x > @y
PRINT 'x y\'den büyüktür';
ELSE
PRINT 'x y\'den küçük veya eşittir';
-- bana burda hata vermeyecek
```

CASE KULLANIMI;

koşullu değer döndürür

```
--1)SORGU İÇİNDE KULLANIMI
SELECT
OgrenciAd,
Yas,
CASE
WHEN Yas < 18 THEN 'Çocuk'
WHEN Yas BETWEEN 18 AND 25 THEN 'Genç'
ELSE 'Yetişkin'
END AS YasGrubu
FROM Ogrenciler;
/* ogrenciler tablomdan ogrenciad , yas 'ı ve
yasgrubu adı altında verilen koşula göre yazılmasını istiyorum. yani
ogrenciAd  yas  yasgrubu
ahmet      23  yetişkin
ali        12  çocuk
*/
```

```
--2) DEĞİŞKEN İÇİNDE
DECLARE @puan INT = 75;
SELECT
CASE
WHEN @puan >= 90 THEN 'AA'
WHEN @puan >= 80 THEN 'BA'
WHEN @puan >= 70 THEN 'BB'
ELSE 'Geçmedi'
END AS NotDurumu;
/* NotDurumu ismi altında koşula göre olanı yazdırıyor. */
```

WHILE DÖNGÜSÜ

belli bir koşul sağlandığı sürece tekrarlanır.

```
WHILE (koşul)
BEGIN
-- kod bloğu
END

--1DEN 5E KADAR SAYILARI YAZDIRMA
```

```
DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
    PRINT 'Sayi: ' + CAST(@i AS VARCHAR);
    SET @i = @i + 1;
END
```

BREAK VE CONTINUE

break: döngüden çıkar

continue: içinde olduğu turu atlar ve bir sonraki tura geçer

```
DECLARE @i INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 3
        CONTINUE; -- 3 yazılmaz
    IF @i = 7
        BREAK; -- Döngü biter
    PRINT 'Sayi: ' + CAST(@i AS VARCHAR);
END
/* i adında tanımladığım değişken döngünün dönmesini sağlayan sayaç
i 10dan küçük olduğu süre boyunca döngü dönsün.
cast ile int tipinde olan değişkenimi bastırırken
varchar haline dönüştürüyorum.
break kullanarak döngünün 7 olduğu zaman bitmesini sağlıyorum.
continue kullandığım 3ü atlayarak döngünün devam etmesini sağlıyorum.
bir sonraki döngüye devam ediyor.*/
```

KISACA;

DECLARE: değişken tanımlanırken kullanılır.

SET : değişkene değer atadığım zaman.

IF...ELSE : koşula göre karar verir.

CASE: tek satırda çoklu if durumunu sağlar

WHILE : döngü kurar

USE VE GO KULLANIMI

USE;


```
USE VeritabaniAdi;  
USE okulDB;  
select*from ogrenciler;  
-- okulDB database kullanarak ogrenciler tablosunu görüyorum.
```

GO;

Batch = Komut Grubu

SQL Server komutları parça parça işler.

GO yazmazsan, her şey tek parça (tek batch) olarak işlenir.

SQL Server bazı komutları **gruplar (batch) halinde** çalıştırmak ister.

Örneğin:

- Değişken tanımlama (**DECLARE**) ve kullanımı aynı grupta olmalı.
- Yeni bir tablo oluşturduktan sonra hemen veri ekleyeceksen, araya **GO** koymak gerekebilir.

Yani SQL Server'a "**Buraya kadar olan komutları bir grup (batch) olarak çalıştır**" demek için kullanılır.

- Her **GO** bir bölümdür.
- Komutlar birbirinden ayrıdır ama doğru yerlerde kullanılırsa sorunsuz çalışır.

komutları parçalara ayırır. değişken tanımlama gibi işlemleri GO ile ayrılmazsa hata alabilirsin.

Sen **GO** yazınca, SSMS şunu yapar:

- "Buraya kadar olanları al → SQL Server'a gönder."
- Sonra kalan kısmı **ayrı bir işlem gibi** başlatır.

GO sonrası değişken kullanılamaz

Değişken tanımlayıp **sonra** kullanacaksan

☒ Evet

Farklı işlemleri ayrı gruplarda çalıştıracaksan

☒ Evet

```
USE OkulDB;  
GO  
  
DECLARE @ad VARCHAR(20) = 'Ahmet';  
SELECT @ad AS Isim;  
GO
```

```
DECLARE @sayac INT = 1;  
GO
```

```
WHILE @sayac <= 3  
BEGIN  
    PRINT 'Sayac: ' + CAST(@sayac AS VARCHAR);  
    SET @sayac = @sayac + 1;  
END
```

/*BURDA HATA VERİR ÇÜNKÜ GO KULLANIMINDAN SONRA TANIMLANAN SAYAC DEĞİŞKENİ KAYBOLUYOR.TEKRAR KULLANILAMIYOR.*/

İNDEKS KAVRAMI VE ÇEŞİTLERİ

İNDEKS: veri tabanında aramaları hızlandırmak için kullanılıyor. bir tabloda binlerce satır varsa sql tek tek kontrol etmek yerine index bakarak hızlıca ulaşıyor. sorgu performansı where join orderby varsa ciddi derecede artıyor.

AMA: her veri ekleme silme ve güncelleme işlemleri indexte güncellenir. yazma işlemleri biraz yavaşlar. Çok fazla index olursa sistem şişebilir.

temel index türleri:

clustered index: gerçek veriyi sıralar ve saklar sadece 1 tane olabilir.

verinin fiziksel sıralamasını belirler. tabloda sadece 1 tane olabilir. genelde PK sütununda otomatik olarak oluşur.

örnek bir tablo üzerinden gidelim ilk başta

OgrenciID	Ad	Soyad	Durum	TCKimlikNo	Icerik
1	Zeynep	Kaya	Aktif	12345678901	Veritabanı dersini çok...
2	Ali	Demir	Pasif	12345678902	SQL Server öğrenmek...
3	Ayşe	Yılmaz	Aktif	12345678903	Index performansı artırır.
4	Mehmet	Can	Aktif	12345678904	Full-text arama gücü...

```
select* from Ogrenciler WHERE Ad="Zeynep";  
CREATE CLUSTERED INDEX idx_ogrenci_id  
ON Ogrenciler(OgrenciID);
```

burada veriler ogrenciID ye göre sıralı şekilde gelir. çünkü

non-clustered index : gerçek veri ayrı,index ayrı tutulur. bir tabloda birden fazla olabilir.
Non-Clustered Index = veriyi fiziksel olarak değil,
ayrı bir alanda indeks olarak tutar. Özellikle arama işlemlerini hızlandırmak için kullanılır
(örneğin: `WHERE Ad = 'Zeynep'` gibi sorgular).

idx_ogrenci_ad: oluşturulan indexe verilen isimdir. sqlde indexe özel bir isim vermek zorundasın. bu isim sistemde indexi tanımlamak için kullanılır.

on ogrenciler (ad) : ogrenciler tablosundaki ad sütununa uygula demek

```
SELECT * FROM Ogrenciler WHERE Ad = 'Zeynep';  
--burda satırları sql tek tek tarar ve yavaştır.  
CREATE NONCLUSTERED INDEX idx_ogrenci_ad ON Ogrenciler(Ad);  
--sorgumu yaparım  
-- Daha hızlı çalışır, çünkü Ad sütununa indeks ekledik  
SELECT * FROM Ogrenciler WHERE Ad = 'Zeynep';
```

unique index : aynı değeri iki kere almayı engeller benzersiz alan içindir.

```
CREATE UNIQUE INDEX idx_tc ON Ogrenciler(TCKimlikNo);  
-- tckimlik benzersiz olacak,  
  
INSERT INTO Ogrenciler VALUES  
(5, 'Ahmet', 'Toprak', 'Aktif', '12345678901', 'Tekrar veri');  
--tekrar edildiğinde ise hata verecek.
```

filtered index : belirli koşula uyan veriler için index oluşturur.

avantajı ise index sadece aktif olan öğrencileri içerdiği için daha az yer kaplar bu yüzden daha hızlı olacak.

```
-- yalnızca durum=aktif olan öğrenciler için geçerli olacak  
CREATE NONCLUSTERED INDEX idx_aktif_ogrenci  
ON Ogrenciler(Ad)  
WHERE Durum = 'Aktif';  
  
--sorgu  
SELECT * FROM Ogrenciler  
WHERE Durum = 'Aktif' AND Ad = 'Ayşe';
```

OgrenciID	Ad	Soyad	Durum
3	Ayşe	Yılmaz	Aktif

full-text index : metin içinde kelime aramak için kullanılır.(like yerine daha güçlü)

içerik sütununda kelime aramak

```
-- Full-text index tanımı yapılmış varsayalım
SELECT * FROM Ogrenciler
WHERE CONTAINS(Icerik, 'veritabanı');

--!!! bu ifade LIKE'% veritabanı%' ifadesinden daha doğru ve hızlı çalışır.
```

bu işlem index'i yeniden düzenler (fragmentation düşer) görsel çıktısı yok ama sorgular hızlanabilir.

```
ALTER INDEX idx_ogrenci_ad ON Ogrenciler REBUILD;
```

ÖRNEK 1 :

Senaryo: Bir öğretmen öğrencilerin isimlerini alıp sisteme bakmak istiyor. Sadece AKTİF durumdaki öğrencilerle ilgileniyor. Eğer öğrenci bulunmazsa ÖĞRENCİ BULUNAMADI yazılacak. AD sütununda index var.

adı zeynep ve aktifse gösterecek, pasif yada yoksa bulunamadı çıktısı verecek.

```
CREATE NONCLUSTERED INDEX idx_ogrenci_ad ON ogrenciler(ad);
--index oluşturma yaptım.
DECLARE @ad varchar(20)= 'zeynep';
--değişken tanımı
IF EXISTS( select 1 FROM ogrenciler WHERE ad=@ad and durum='aktif')
begin
    select *from ogrenciler where AD="@ad durum= 'aktif';
end
else
begin
    print'öğrenci bulunamadı';
end
```

ÖRNEK 2 :

Öğrenci sistemi kullanan bir öğretmen, girilen **öğrenci adı** için sorgu yapmak istiyor. Eğer öğrenci:

- Aktifse → bilgilerini ve **kayıt uzunluğunu, isim uzunluğunu, kayıt yılına +1 yıl** eklenmiş tarihi göstereceğiz.

- Pasifse → "Pasif öğrenci" mesajı
- Hiç yoksa → "Öğrenci bulunamadı" yazacağız.

```
-- 1.aşama index oluşturma
create nonclustered index idx_ad on ogrenciler(ad);,
go

--2. aşama değişken tanımları yapma
declare @ varchar(50)= 'zeynep';
declare @ogrenciid int;
declare @durum varchar(10);
declare @isimuzunlugu int;
declare @yenitarih date;

--3.aşama öğrenci var mı kontrol
if exist( select 1 from ogrenciler where ad=@ad)
begin
    --değerlerin alınması
    select top 1
        @ogrenciid=ogrenciid,
        @durum= durum,
        @isimuzunlugu=len(ad)
    from ogrenciler where ad=@ad;

    if @durum='aktif'
    begin
        set @yenitarih = dateadd(year,1,getdate())
        select ogrenciid,ad,soyad,durum, @isimuzunlugu as IsimUzunluğu,
            substring(ad,1,3) as Isimilk3Harfi,
            format(@yenitarih, 'yyyy-MM-dd') as yenitarih;
    end
else
    begin
        print'öğrenci pasif';
    end
end
else
    begin
        print 'öğrenci bulunamadı';
    end
end

/*
!zeynep adında aktif bir öğrenci varsa;
```

OgrenciID	Ad	Soyad	Durum	IsimUzunlugu	IsminIlk3Harfi	YeniTarih
1	Zeynep	Kaya	Aktif	6	Zey	2026-04-07

!zeynep pasifse;
öğrenci pasif

!zeynep yoksa;
öğrenci bulunamadı olarak çıktı verecek

*/

FONKSİYON OLUŞTURMA:

fonksiyon sqlde bir veya daha fazla parametre alarak işlem yapar ve sonuç döndürür.

fonksiyon türleri

1. built-in functions
2. user defined functions(udf)

Built-in func(hazır fonksiyonlar):

len() metin uzunluğunu döndürür → len('zeynep') → 6

getdate() şuanki tarih saat döndürür → getdate()

isnull(a,b) null yerine başka değer döndürür → isnull(null,'boş') → 'boş'

substring() metinden parça alır → substring('ahmet', 2,2) >hm

dateadd() tarihe süre ekler → dateadd(day,5,getdate())

user-defined func (kullanıcı tanımlı fonk):

bunları kendimiz oluştururuz ve tekrar tekrar kullanılabilir.

scalar function → tek bir değer döner (örn: yas hesaplama)

table-valued func → tablo döner(sorgu sonucu gibi)

1.scalar function: tek değer döndüren fonk

bir öğrencinin doğum yılını hesaplayalım

```
create function fn_yashesaplama(@dogumyili int)
return int as
begin
    declare @yas int;
    set @yas= year(getdate())-@dogumyili;
    return @yas;
```

```

end;

--sorguyu hazırlarken
SELECT dbo.fn_YasHesapla(2003) AS Yas;

Yas
---
21
-- olarak da çıktı verir.

```

2.table-valued function(tablo döndüren fonksiyon)

ogrenciler tablosundaki sadece aktif olanları listele

```

create function fn_aktifogrenciler()
returns table as
return(
    select *from ogrenciler where durum="aktif ");

SELECT * FROM dbo.fn_AktifOgrenciler();
/*
create function→ yeni fonksiyon oluşturur.
returns fonksiyonun ne döndüreceğini belirtir
return döndürdüğü değeri belirtir.(scalar için)
dbo. fonksiyonun şema adı(genelde dbo)
select*from dbo.fn() tablo döndüren fonksiyonlar böyle çağırılır.
*/

--çıktısı aşağıdaki gibi olur.

```

OgrenciID	Ad	Soyad	Durum
1	Zeynep	Kaya	Aktif
3	Ayşe	Yılmaz	Aktif
...

örnek1:

verilen ad ve soyad değerlerini birleştirip tam adı döndüren bir fonksiyon.

```

create function fn_tamad(@ad varchar(50),@soyad varchar(50))
return varchar(100) as
begin
    return @ad + ' ' + @soyad;

```

```

end;

SELECT dbo.fn_TamAd('Zeynep', 'Kaya') AS TamAd;

--çıktısı
--tamad
--zeynep kaya

```

örnek 2:

sınav puanını girince karşılığında harf notu döndüren fonk

```

create function fn_harfnotu(@puan int)
return varchar(2) as
begin declare @not varchar(2);
    set @not=
        case
            when @puan >= 90 then 'aa'
            when @puan >= 80 then 'ba'
            when @puan >= 70 then 'bb'
            when @puan >= 60 then 'cb'
            else 'ff'
        end;
    return @not;
end;

select dbo.fn_harfnotu(75) as hafnotu;

--harfnotu
--bb

```

örnek3: tablevalued func

belli bölümdeki öğrenciler, girilen bölüm adına göre o bölüme ait öğrencileri döndürsün.

```

create funtion fn_bölüme göre öğrenciler(@bolum varchar(50))
return table as
return(
    select *from ogrenciler
    where bolum @bolum
);
-- bu örnek için tablonun boluum adında bir sütün olduğunu varsayıyoruz.
select *from dbo.fn_bolumegöre öğrenciler ('bilgisayar mühendisliği');

```


OgrenciID	Ad	Soyad	Bolum
5	Ali	Yılmaz	Bilgisayar Mühendisliği
7	Zeynep	Koç	Bilgisayar Mühendisliği

örnek4: tablo örneği

doğum günü bugün olan öğrencileri listele.

```
create function fn_bugundugumluogrenciler()
returns table
as
return (
    select *from ogrenciler
    where day(dogumtarihi)=day(getdate()) and
    month (dogumtarihi)= month(getdate())
);
-- tabloya göre dgumtarihi sütunu eklenmiş olmalı

select* from dbo.fn_bugundogumluogrenciler();
```

örnek5: kısa ad oluşturma

verilen ad ve soyadın baş harflerini alarak kısa isim üretmek

```
create function fn_kisaad(@ad varchar(50) ,@soyad varchar(50))
returns varchar(10) as
begin
    return left(@ad,1)+'.'+ left(@soyad,1)+'.';
end;

select dbo_fnkisaad('zeynep','kaya') as kisaad;

--kisaad
z.k.
```

örnek6: epostaoluşturucu

öğrencinin adı soyadı ve öğrenci numarasına göre eposta oluşturan

```
create function fn_epostaolustur(
@ad varchar(50),
@soayad varchar(50),
```

```

@no int
)
returns varchar(100)
as
begin
    return lower(@ad + '.' + @soyad + cast( @no as varchar))+'@okul.edu.tr';
end;

select dbo.fn_epostaolustur('zeynep','kaya',123) as eposta;

--zeynep.kaya123@okul.edu.tr

```

örnek7: belli harfle başlayan öğrenciler

```

create function fn_adiilebaslayan(@harf char(1))
returns table as
return (
    select *from ogrenciler
    where ad like @harf + '%');

select *from dbo.fn_adiilebaslayan('A');

```

OgrenciID	Ad	Soyad
2	Ayşe	Yılmaz
4	Ali	Demir

ÖRNEK8: telefon formatlayıcı

11 hane girilen tel nosunu +90xxxxxxxxxx formatına dönüştür

```

create function fn_telefonformat(@telno char(11))
returns varchar(20) as
begin return '+90'+
    SUBSTRING(@telNo, 1, 3) + '-' +
    SUBSTRING(@telNo, 4, 3) + '-' +
    SUBSTRING(@telNo, 7, 2) + '-' +
    SUBSTRING(@telNo, 9, 2);
end;
SELECT dbo.fn_TelefonFormatla('5381234567') AS FormatliTel;

```

FormatliTel

+90 538 123 45 67

VIEW KULLANIMI :

sorguların sadeleşmesi, güvenlik , yeniden kullanılabilirlik ve geçici işlemler için
view bir yada daha fazla tablodan veri sorgulayan sanal tablodur. gerçekte veri içermez.

```
create view view_adsoyad as select ad,soyad from ogrenciler;  
-- bu komutla view_adsoyad adında sanal tablo oluşturduk.  
select* from view_adsoyad;
```

Amaç	Açıklama
Sorguyu sadeleştirmek	Karmaşık sorgular için yeniden kullanılabilir
Güvenlik	Kullanıcıyı sadece gerekli sütunlara erişirme
Tekrar kullanılabilirlik	Aynı SELECT sorgusunu tekrar yazmana gerek kalmaz

```
create view view_aktifogrenciler as select*from ogrenciler where durum='aktif';  
select* from view_aktifogrenciler;  
--arkada where koşunu sağlayanları alıyor.
```

view güncelleme ve silme işlemleri;

```
--güncelleme  
alter view view_soyad as select ad,soyad,bolum,from ogrenciler;
```

```
--silme  
drop view view_adsoyad;
```

geçici tablolar sorgu sırasında geçici olarak oluşturulan ve oturum kapanınca silinen tablolardır.
tek # → oturuma özel geçici tablolar

```
create table #geciciogrenciler(  
ogrenciID int, ad varchar(50));  
-- bu tablo sadece benim oturumum boyunca geçerli. başka kullanıcı göremez.
```

```
--kullanımına gelecek olursak da
insert into #geçiciogrenciler values(1,'ali'),(2,'ayşe');
select *from #geçiciogrenciler
```

çift ## → tüm oturumlara açık geçici tablo

```
create table ##genelgeçicitable(
ad varchar(50));
-- bu tabloda tüm kullanıcılar erişim sağlıyor.
```

Özellik	View	Geçici Tablo
Veri içerir mi?	✗ Hayır, sadece sorgudur	✓ Evet, gerçek veriyi tutar
Ne zaman oluşur?	Kalıcıdır (silene kadar durur)	Oturum boyunca geçerlidir
Kullanım amacı	Sorguyu sadeleştirme	Geçici işlemler, filtreleme, test
Silinme şekli	DROP VIEW	Oturum kapanınca otomatik silinir

```
-- VIEW oluştur
CREATE VIEW view_AktifBilgisayar
AS
SELECT Ad, Bolum FROM Ogrenciler WHERE Durum = 'Aktif' AND Bolum = 'Bilgisayar';

-- GEÇİCİ TABLO oluştur
CREATE TABLE #GeciciYedek (OgrenciID INT, Ad VARCHAR(50));

INSERT INTO #GeciciYedek SELECT OgrenciID, Ad FROM Ogrenciler WHERE Durum = 'Pasif';
```

örnek1: basit view oluşturma

```
--OGRENCİLER TABLOSU
CREATE TABLE Ogrenciler (
  OgrenciID INT PRIMARY KEY,
  Ad VARCHAR(50),
  Soyad VARCHAR(50),
  Bolum VARCHAR(50),
  Durum VARCHAR(10)
);
--NOTLAR TABLOSU
```

```
CREATE TABLE Notlar (
  OgrenciID INT,
  Ders VARCHAR(50),
  Puan INT
);
```

```
create view view_aktifogrenciler as
select ogrenciID,ad,soyad,bolum
from ogrenciler where durum='aktif';
select *from view_aktifogrenciler;
```

OgrenciID	Ad	Soyad	Bolum
1	Ali	Demir	Bilgisayar
2	Zeynep	Kaya	Endüstri

örnek2: join ile view: öğrenci+notlar

öğrencilerin adı soyadı ile birlikte aldığı ders ve puanı göster

```
create view view_ogrencinotlari as
select
  o.ad+' '+ o.soyad as tamad,
  o.bolum,
  n.ders
  n.puan
from ogrenciler as o join notlar as n on o.ogrenciID= n.ogrenciID;

select* from view_ogrenciNotlari;
```

TamAd	Bolum	Ders	Puan
Ali Demir	Bilgisayar	Veritabanı	85
Zeynep Kaya	Endüstri	Algoritma	72

örnek3: groupby ile kullanımı bölümlere göre ortalama

```
create view view_bolumortalama as
select o.bolum,
avg(n.Puan) as ortalamaPuan from ogrenciler o join notlar n on o.ogrencilerID=n.ogrenciID
group by o.bolum;
```

```
select * from view_bolumortalama;
```

Bolum	OrtalamaPuan
Bilgisayar	84.5
Endüstri	76.0

örnekl4: sadece 80 üzeri notlar

```
CREATE VIEW view_BasariliOgrenciler  
AS  
SELECT o.Ad, o.Soyad, n.Ders, n.Puan  
FROM Ogrenciler o  
JOIN Notlar n ON o.OgrenciID = n.OgrenciID  
WHERE n.Puan >= 80;  
  
SELECT * FROM view_BasariliOgrenciler;
```

Ad	Soyad	Ders	Puan
Ali	Demir	Veritabanı	85
Zeynep	Kaya	Programlama	92

JOINLER;

1 Öğrenciler Tablosu:

OğrenciID	Ad
1	Ali
2	Ayşe
3	Zeynep

2 Notlar Tablosu:

OğrenciID	Ders	Puan
1	Mat	85
2	Türkçe	70
4	Fen	90

◆ 1. INNER JOIN

sql

Kopyala

Düzenle

```
SELECT o.Ad, n.Ders, n.Puan
FROM Öğrenciler o
INNER JOIN Notlar n ON o.OğrenciID = n.OğrenciID;
```

● Sadece eşleşen kayıtları getirir:

Ad	Ders	Puan
Ali	Mat	85
Ayşe	Türkçe	70

◆ 2. LEFT JOIN

sql

📋 Kopyala

✎ Düzenle

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
LEFT JOIN Notlar n ON o.OgrenciID = n.OgrenciID;
```

● Sol (Ogrenciler) tüm kayıtları alır, eşleşmeyen Notlar'a NULL yazar:

Ad	Ders	Puan
Ali	Mat	85
Ayşe	Türkçe	70
Zeynep	NULL	NULL

◆ 3. RIGHT JOIN

sql

📋 Kopyala

✎ Düzenle

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
RIGHT JOIN Notlar n ON o.OgrenciID = n.OgrenciID;
```

● Sağ (Notlar) tüm kayıtları alır, eşleşmeyen öğrencilere NULL yazar:

Ad	Ders	Puan
Ali	Mat	85
Ayşe	Türkçe	70
NULL	Fen	90

◆ 4. FULL OUTER JOIN

sql

Kopyala

Düzenle

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
FULL OUTER JOIN Notlar n ON o.OgrenciID = n.OgrenciID;
```

● Her iki tablodan tüm kayıtları alır, eşleşmeyene NULL yazar:

Ad	Ders	Puan
Ali	Mat	85
Ayşe	Türkçe	70
Zeynep	NULL	NULL
NULL	Fen	90

İhtiyaç

JOIN Türü

Her iki tabloda da olanları istiyorum

INNER JOIN

Soldakilerin hepsi, sağda varsa

LEFT JOIN

Sağdakilerin hepsi, solda varsa

RIGHT JOIN

Her şeyi, ne varsa getir

FULL OUTER JOIN

Kombinasyonları üretmek istiyorum

CROSS JOIN

ÖRNEK: joinleriçin

bu şuan örnek tablo

```
-- Öğrenciler tablosu
CREATE TABLE Ogrenciler (
  OgrenciID INT,
  Ad VARCHAR(50)
```

```
);

INSERT INTO Ogrenciler VALUES
(1, 'Ali'),
(2, 'Ayşe'),
(3, 'Zeynep');

-- Notlar tablosu
CREATE TABLE Notlar (
    OgranciID INT,
    Ders VARCHAR(50),
    Puan INT
);

INSERT INTO Notlar VALUES
(1, 'Matematik', 85),
(2, 'Türkçe', 70),
(4, 'Fizik', 90); -- bu öğrenci Ogrenciler tablosunda yok
```

INNER JOIN KULLANIMI

```
--INNER İÇİN KULLANIMI;

SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
INNER JOIN Notlar n ON o.OgranciID = n.OgranciID;
```

Ad	Ders	Puan
Ali	Matematik	85
Ayşe	Türkçe	70

LEFT JOIN – Soldaki tüm öğrenciler, not varsa ekle

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
LEFT JOIN Notlar n ON o.OgranciID = n.OgranciID;
```

Ad	Ders	Puan
Ali	Matematik	85
Ayşe	Türkçe	70
Zeynep	NULL	NULL

Zeynep'in notu yok ama Ogrenciler tablosunda var → NULL'la döner

OgrenciID = 4 notu → Ogrenciler tablosunda olmadığı için gelmez

RIGHT JOIN – Notlar tablosundaki her şey gelsin

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
RIGHT JOIN Notlar n ON o.OgrenciID = n.OgrenciID;
```

Ad	Ders	Puan
Ali	Matematik	85
Ayşe	Türkçe	70
NULL	Fizik	90

FULL OUTER JOIN – Tüm kayıtlar gelir

```
SELECT o.Ad, n.Ders, n.Puan
FROM Ogrenciler o
FULL OUTER JOIN Notlar n ON o.OgrenciID = n.OgrenciID;
```

Ad	Ders	Puan
Ali	Matematik	85
Ayşe	Türkçe	70
Zeynep	NULL	NULL
NULL	Fizik	90

ÖRNEK İÇİN TABLOLAR

Oğrenciler Tablosu

	OğrenciID	Ad	Soyad	Bolum	Durum	DogumTarihi
1	1	Ali	Demir	Bilgisayar	Aktif	2003-05-15
2	2	Ayşe	Kaya	Endüstri	Aktif	2002-08-21
3	3	Zeynep	Yılmaz	Bilgisayar	Pasif	2004-12-10
4	4	Mehmet	Aslan	Makine	Aktif	2001-11-02

Notlar Tablosu

	OğrenciID	Ders	Puan
1	1	Veritabanı	85
2	1	Algoritma	75
3	2	Matematik	90
4	2	Türkçe	70
5	4	Fizik	65
6	5	Kimya	88

ÖRNEK: "BAŞARILI ÖĞRENCİLERİ LİSTELE"

Bir okul yönetimi, sadece "Aktif" durumdaki ve notu 70'in üzerinde olan öğrencilerin tam adlarını ve not durumlarını görmek istiyor

--FONKSİYON HARF NOTU HESAPLA

```
CREATE FUNCTION fn_HarfNotu (@puan INT)
RETURNS VARCHAR(2)
AS
BEGIN
    RETURN
    CASE
        WHEN @puan >= 90 THEN 'AA'
        WHEN @puan >= 80 THEN 'BA'
        WHEN @puan >= 70 THEN 'BB'
        ELSE 'FF'
    END;
END;
```

```

END;

--VIEW AKTİF VE BAŞARILI ÖĞRENCİLERİ GÖSTERİYİRUM
CREATE VIEW view_BasariliOgrenciler
AS
SELECT
    o.Ad + ' ' + o.Soyad AS TamAd,
    n.Ders,
    n.Puan,
    dbo.fn_HarfNotu(n.Puan) AS HarfNotu
FROM Ogrenciler o
JOIN Notlar n ON o.OgrenciID = n.OgrenciID
WHERE o.Durum = 'Aktif' AND n.Puan >= 70;

--KULLANIMINI İSE
SELECT * FROM view_BasariliOgrenciler;

```

ÖRNEK2: "ÖĞRENCİ E-POSTA VE YAŞ HESABI"

E-Posta sisteminde öğrencinin adı, soyadı ve numarasından otomatik e-posta oluşturulmalı. Aynı zamanda yaş bilgisi de listelenmeli.

```

--Fonksiyon: E-posta oluştur

CREATE FUNCTION fn_EpostaOlustur (
    @ad VARCHAR(50), @soyad VARCHAR(50), @no INT
)
RETURNS VARCHAR(100)
AS
BEGIN
    RETURN LOWER(@ad + '.' + @soyad + CAST(@no AS VARCHAR)) + '@okul.edu.tr';
END;

--FONKSİYON YAŞ HESAPLAMA
CREATE FUNCTION fn_Yas (@dogum DATE)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(YEAR, @dogum, GETDATE());
END;

--DORGUSU

```

```
SELECT
    Ad, Soyad,
    dbo.fn_EpostaOlustur(Ad, Soyad, OgrenciID) AS Eposta,
    FORMAT(DogumTarihi, 'yyyy-MM-dd') AS Dogum,
    dbo.fn_Yas(DogumTarihi) AS Yas
FROM Ogrenciler;
```

ÖRNEK 3: "Bölüm Bazında Not Ortalaması"

Her bölüm için öğrencilerin ortalama notunu listele. Sadece ortalaması 70'in üzerinde olan bölümler gelsin.

```
--Index ekle (performans için)

CREATE NONCLUSTERED INDEX idx_OgrenciBolum ON Ogrenciler(Bolum);

--VIEW OLUŞTURMA
CREATE VIEW view_BolumOrtalamasi
AS
SELECT
    o.Bolum,
    AVG(n.Puan) AS Ortalama
FROM Ogrenciler o
JOIN Notlar n ON o.OgrenciID = n.OgrenciID
GROUP BY o.Bolum;



--DORGUSU

SELECT * FROM view_BolumOrtalamasi
WHERE Ortalama > 70;
```

soru 1:

MsSQL'de kullanılan Geçici Tablolar (Temporary Tables) nelerdir? Hangi amaçla kullanılmaktadır? Geçici tablo tipleri nelerdir ve bu tipler arasındaki farklar nelerdir?

 Bu soruda şunları açıklamalısın:

- Geçici tablo nedir? → Oturum boyunca çalışan, geçici veri tutan tablolardır.
- Tipleri:
 -  → Oturuma özel
 -  → Global/geçici ama tüm oturumlara açık

- Kullanım amaçları:
 - Geçici hesaplama saklamak
 - Karmaşık sorguları bölmek
 - Geçici veri depolamak
- Fark:
 - #tablo sadece **senin oturumunda**
 - ##tablo herkes için erişilebilir

soru2:

✓ 2. Soru: Sistem Veritabanları (10 Puan)

MsSQL'de bulunan sistem veritabanlarını ve bu veritabanlarının kullanım amaçlarını açıklayınız.

💡 Burada açıklaman gereken veritabanları:

Sistem Veritabanı	Açıklama
master	Tüm sistem ayarları, login bilgileri
model	Yeni veritabanı şablonu
msdb	SQL Server Agent görevleri
tempdb	Geçici tablolar, işlemler

soru3:

Table Scan nedir?

- SQL Server, tabloyu **başlangıçtan sonuna kadar okur**
- Her bir satırı tek tek inceler: "Bu satır aranan değeri içeriyor mu?"

Durum	SQL Server Ne Yapar?
Hiç index yok	Table Scan yapar
WHERE / JOIN / ORDER BY varsa	Yine tüm tabloyu satır satır tarar
Büyük tabloda ciddi performans kaybı	Evet

! SQL Server, tüm tabloyu satır satır tarar. Buna Table Scan (tam tablo taraması) denir.

Durum	Sonuç
Index yok	Tüm tablo taranır (Table Scan)
Sorgular yavaş çalışır	Özellikle büyük tabloda belirgin
CPU ve IO maliyeti artar	Sistem yavaşlar
Execution Plan kötüleşir	Optimization zorlaşır

Index yoksa:

- SQL Server bütün tabloyu tarar.
- Her satıra bakar: "Bu satırdaki Ad 'Zeynep' mi?"



Bu işlem:

- Yavaş
- CPU ve disk yükü fazla
- Özellikle büyük tablolarda ciddi performans kaybı

♦ 1. Index Seek (NonClustered) neden oluştu?

`FirstName = 'Ken'` koşulu geldiği için, SQL Server `FirstName` sütununa tanımlı bir Non-Clustered Index kullanarak doğrudan arama yaptı.

→ Bu iyi bir şeydir. Sadece gerekli satırlara gidilir, tablo taranmaz.

♦ 2. Key Lookup (Clustered) neden oluştu?

`SELECT` ifadesinde sadece `FirstName` yok, aynı zamanda `LastName`, `PersonType`, `BusinessEntityID` de istendi.

Ama Non-Clustered Index sadece `FirstName` içeriyor.

💡 Bu nedenle SQL Server:

İlk olarak Index Seek ile `FirstName` değerini buldu, sonra Clustered Index'e dönüp diğer sütunları aldı = Key Lookup

♦ 3. Nested Loops neden oluştu?

Çünkü SQL Server, küçük veri setlerinde Index Seek ile eşleşenleri bulur, sonra her biri için tek tek Clustered Index'e giderek detayları alır.

Bu, genellikle küçük veya filtreli sorgularda kullanılır.

soru4:

Aşağıda yapısı verilen `city` tablosunda:

- `city_name` (varchar),
- `lat` (float),
- `long` (float),
- `country_id` (int),
- `population` (int)

Londra'nın uzunluk (`long`) bilgisi diğer şehirlerle karşılaştırılarak bir fonksiyon yazılması isteniyor.

Fonksiyon, bir şehrin **Londra'nın doğusunda mı batısında mı** olduğunu belirtmelidir.

Fonksiyon adı: `east_or_west`

Girdi: `@city_long`

Dönüş: `'east'` veya `'west'`

```
CREATE FUNCTION east_or_west (@city_long FLOAT)
RETURNS VARCHAR(10)
AS
BEGIN
    DECLARE @result VARCHAR(10);

    IF @city_long > -0.1278
        SET @result = 'east';
    ELSE
        SET @result = 'west';

    RETURN @result;
END;

SELECT dbo.east_or_west(30.5) AS Konum1; -- İstanbul → 'east'
SELECT dbo.east_or_west(-74.0) AS Konum2; -- New York → 'west'
```

Açıklama:

- Fonksiyon `@city_long` değerini alır
- Sabit Londra boylamı ile karşılaştırır
- Koşula göre `'east'` ya da `'west'` döndürür

iyileştirilmiş fonksiyon

```
CREATE FUNCTION east_or_west (@city_long FLOAT)
RETURNS VARCHAR(10)
AS
BEGIN
    RETURN
    CASE
        WHEN @city_long > -0.1278 THEN 'east'
        ELSE 'west'
    END;
END;
```

soru5:

Personel tablosunda **ID** (int), **Name** (varchar), **Surname** (varchar), **Gender** (varchar), **BirthDate** (datetime) ve **DepartmentID** (int) alanları bulunmaktadır.

Department tablosunda ise **DepartmentID** (int) ve **DepartmentName** (varchar) alanı bulunmaktadır.

Bu iki veritabanı yöneticisi şirketteki çalışanların tüm bilgilerini **departman adıyla birlikte** listelemek istemektedir.

Bu işi sürekli kod yazarak tekrar etmek yerine, bu işi yapan bir sanal tablo (VIEW) oluşturmak istemektedirler.

Bu nedenle **"AllEmployeesInfo"** isimli **view**'i aşağıya yazınız:

```
CREATE TABLE Personel (  
    ID INT,  
    Name VARCHAR(50),  
    Surname VARCHAR(50),  
    Gender VARCHAR(1),  
    BirthDate DATE,  
    DepartmentID INT  
);  
  
CREATE TABLE Department (  
    DepartmentID INT,  
    DepartmentName VARCHAR(100)  
);  
  
CREATE VIEW AllEmployeesInfo  
AS  
SELECT  
    p.ID,  
    p.Name,  
    p.Surname,  
    p.Gender,  
    p.BirthDate,  
    d.DepartmentName  
FROM Personel p  
JOIN Department d ON p.DepartmentID = d.DepartmentID;
```

		ID	Name	Surname	Gender	BirthDate	DepartmentID	DepartmentName
1	0	1	Ali	Demir	M	1998-05-12	101	Bilgi İşlem
2	1	3	Ayşe	Yılmaz	F	1999-12-05	101	Bilgi İşlem
3	2	2	Zeynep	Kaya	F	2000-08-23	102	İnsan Kaynakları

SINAVA BENZER SORULAR

SORU1:

Aşağıdaki özellikleri sağlayan bir **scalar function** yazınız:

Fonksiyon adı: `fn_YasKategori`

Girdi: doğum yılı (INT)

Dönüş:

- 0–17 arası → `'Çocuk'`
- 18–25 arası → `'Genç'`
- 26 ve üzeri → `'Yetişkin'`

```
CREATE FUNCTION fn_YasKategori (@dogumtarihi DATE)
RETURNS VARCHAR(50)
AS
BEGIN
    DECLARE @yas INT;
    DECLARE @kategori VARCHAR(50);

    SET @yas = DATEDIFF(YEAR, @dogumtarihi, GETDATE());

    SET @kategori =
        CASE
            WHEN @yas <= 17 THEN 'Çocuk'
            WHEN @yas BETWEEN 18 AND 25 THEN 'Genç'
            WHEN @yas >= 26 THEN 'Yetişkin'
            ELSE 'Geçersiz ifade'
        END;

    RETURN @kategori;
END;

SELECT dbo.fn_YasKategori('2005-05-10') AS Kategori; -- Çocuk
SELECT dbo.fn_YasKategori('2000-01-01') AS Kategori; -- Genç
SELECT dbo.fn_YasKategori('1985-01-01') AS Kategori; -- Yetişkin
```

soru2:

```
create view OgrenciDersBilgisi as select o.Ad,o.Soyad,n.Ders
from ogrenci o join notlar n ON o.OgrenciID = n.OgrenciID;

SELECT * FROM OgrenciDersBilgisi;
```

soru4: scalar funtion yazımı

Aşağıda yapısı verilen `city` tablosunda:

- `city_name` (varchar),
- `lat` (float),
- `long` (float),
- `country_id` (int),
- `population` (int)

Amaç:

Verilen bir şehrin **Londra'nın doğusunda mı batısında mı** olduğunu belirten bir **scalar function** yazınız.

Londra'nın uzunluk (`long`) değeri `-0.1278` olarak kabul edilecektir.

Fonksiyon adı: `east_or_west`

Girdi: `@city_long` (FLOAT)

Dönüş değeri: `'east'` veya `'west'`

```
CREATE FUNCTION east_or_west (@city_long FLOAT)
RETURNS VARCHAR(10)
AS
BEGIN
    DECLARE @konum VARCHAR(10);

    IF @city_long > -0.1278
        SET @konum = 'east';
    ELSE
        SET @konum = 'west';

    RETURN @konum;
END;
```

haftıçı olduğunu söyleyen

```
CREATE FUNCTION fn_GunKategori (@tarih DATE)
RETURNS VARCHAR(15)
AS
BEGIN
    DECLARE @kategori VARCHAR(15);

    IF DATENAME(WEEKDAY, @tarih) IN ('Saturday', 'Sunday')
        SET @kategori = 'Hafta sonu';
```

```

ELSE
    SET @kategori = 'Hafta içi';

    RETURN @kategori;
END;
SELECT dbo.fn_GunKategori('2024-04-14'); -- Pazar → Hafta sonu

```

kod durumu gösteren

```

CREATE FUNCTION fn_NotDurumu (@puan INT)
RETURNS VARCHAR(10)
AS
BEGIN
    RETURN
    CASE
        WHEN @puan >= 50 THEN 'Geçti'
        ELSE 'Kaldı'
    END;
END;
SELECT dbo.fn_NotDurumu(70); -- Geçti
SELECT dbo.fn_NotDurumu(45); -- Kaldı

```

VİZEDEN SONRASI;

STORE PROCEDURE

store procedure sql serverda daha önce yazılmış ve kaydedilmiş kod blokları olarak geçiyor ve fonksiyonlar gibi çalışıyorlar ama daha güçlü bir yapıya sahiptirler.

içinde insert, update, delete, select, if, while gibi yapılar barındırıyor, parametre alabiliyor ve "exec" komutu ile çalışıyorlar. kullanım amaçlarında kod tekrarını azaltmak, karışık işlemleri bir komut ile çalıştırmak, hızlı olmak ve hata ayıklama işini kolaylaştırmak.

```

--PARAMETRELİ SP KULLANIMI
CREATE PROCEDURE sp_OgrenciGetir
    @ad VARCHAR(50)
AS
BEGIN
    SELECT * FROM Ogrenciler
    WHERE Ad = @ad;
END;

```

```
--çalıştırmak içinde  
EXEC sp_OgrenciGetir @ad='sueda';
```

```
--PARAMETRESİZ SP KULLANIMI  
CREATE PROCEDURE sp_AktifOgrenciler  
AS  
BEGIN  
    SELECT * FROM Ogrenciler WHERE Durum = 'Aktif';  
END;  
-  
EXEC sp_AktifOgrenciler;
```

Stored Procedure ve Function Farkı

Özellik	Function	Stored Procedure
Geri dönüş	Tek bir değer	Geri dönüş zorunlu değil
<code>SELECT</code> içinde kullanma	Evet	Hayır
İçinde <code>INSERT/UPDATE</code>	Hayır	Evet
<code>RETURN</code>	Gerekli	Opsiyonel
<code>EXEC</code> ile çağırma	Hayır	Evet

store procedure ile fonksiyon kullanımı arasındaki farklara gelecek olursak fonksiyonlar;

- **Yan etkisiz (pure)** olmalıdır
- Sadece **hesaplama yapmalı ve değer döndürmeli**
- **Veritabanı içeriğini değiştirmemeli**

Bu yüzden **sadece `SELECT` veya hesaplama işlemleri** yapmana izin verilir.

insert, update, delete, exec, try...catch,raiserror kullanılamaz ama select, if...else, case, declare, set, return, matematik işlemleri, string tarih fonksiyonları kullanımı vardır.

!! eğer veritabanı içeriğini değiştirmek istiyorsam store procedure kullanmalıyım.

Özellik	Function	Stored Procedure
INSERT/UPDATE/DELETE	✗ Yok	✓ Var
Amaç	Hesaplama	İşlem/Yönetim
SELECT içinde	✓ Kullanılır	✗ Kullanılamaz

örnek1:

```

CREATE PROCEDURE sp_OgrenciEkle
    @Ad VARCHAR(50),
    @Soyad VARCHAR(50),
    @DogumTarihi DATE,
    @Bolum VARCHAR(50),
    @Durum VARCHAR(10)
AS
BEGIN
    INSERT INTO Ogrenciler (Ad, Soyad, DogumTarihi, Bolum, Durum)
    VALUES (@Ad, @Soyad, @DogumTarihi, @Bolum, @Durum);

    PRINT 'Öğrenci başarıyla eklendi.';
END;

--çalıştırmak içinde
EXEC sp_OgrenciEkle
    @Ad = 'Zeynep',
    @Soyad = 'Kaya',
    @DogumTarihi = '2002-05-12',
    @Bolum = 'Bilgisayar',
    @Durum = 'Aktif';

```

Satır	Anlamı
CREATE PROCEDURE	Yeni prosedür tanımlıyor
Parametreler (@...)	Dışarıdan alacağı değerler
INSERT INTO ... VALUES	Veriyi tabloya ekliyor
PRINT	Kullanıcıya mesaj döndürüyor

örnek2:

```

CREATE PROCEDURE sp_OgrenciEkle
    @Ad VARCHAR(50),
    @Soyad VARCHAR(50),
    @DogumTarihi DATE,
    @Bolum VARCHAR(50),
    @Durum VARCHAR(10)
AS
BEGIN
    -- Öğrenci zaten var mı? Kontrol et
    IF EXISTS (
        SELECT 1 FROM Ogrenciler
        WHERE Ad = @Ad AND Soyad = @Soyad AND DogumTarihi = @DogumTarihi
    )
    BEGIN
        PRINT 'Bu öğrenci zaten kayıtlı. Ekleme işlemi yapılmadı.';
    END
    ELSE
    BEGIN
        -- Öğrenci yoksa ekle
        INSERT INTO Ogrenciler (Ad, Soyad, DogumTarihi, Bolum, Durum)
        VALUES (@Ad, @Soyad, @DogumTarihi, @Bolum, @Durum);

        PRINT 'Öğrenci başarıyla eklendi.';
    END
END;

--çalıştırmak içi
EXEC sp_OgrenciEkle
    @Ad = 'Zeynep',
    @Soyad = 'Kaya',
    @DogumTarihi = '2002-05-12',
    @Bolum = 'Bilgisayar',
    @Durum = 'Aktif';

```

İf blokları ile öğrenci daha önce var mı diye kontrol ediyoruz else ile kayıtlı değilse ekleme yapıyoruz ve print ile mesaj veriyoruz bu prosedüre göre vtye aynı kişi iki kez eklemeyi önleyen bir güvenlik kontrolü içeriyor. if exist yapısı ile veri bütünlüğü korunmuş oluyor.

peki store procedure neden hızlı?

"Stored Procedure'ler, önceden derlendikleri ve SQL Server plan önbelleğinde saklandıkları için, tekrar çalıştırıldıklarında daha hızlı ve optimize çalışır. Aynı zamanda parametrelili yapıları sayesinde plan tekrar kullanılabilir, bu da performans avantajı sağlar."

Avantaj	Açıklama
Önbellek kullanır	Derlenmiş planı saklar
Daha az trafik	Kod gönderilmez, sadece çağılır
Daha az CPU kullanımı	Tek plan üzerinden çalışır
Yüksek tekrar kullanılabilirlik	Kod tekrarı yok
Güvenlik	Yetkilendirme ile sınırlı kullanım yapılabilir

JOB

belirli işleri otomatik olarak zamanlayıp çalıştırmak için kullanılan yapılardır.

- otomatik çalışan görevlerdir.
- içinde store procedure, backup alma, sorgu çalıştırma, rapor üretme gibi işler olabilir
- arka planda çalışırlar.

📌 Job Özellikleri:

Özellik	Açıklama
Belirli bir zamanlayıcı ile çalışabilir	Örneğin: her gün saat 02:00'de
Tek seferlik veya periyodik olabilir	İstersen her hafta pazartesi
Hata olursa bildirim yapılabilir	E-posta atılabilir, log tutulabilir
Birden fazla adım içerebilir	Örneğin: önce yedek al, sonra logları sil
Başarılı veya başarısız olma durumlarına göre farklı yollar çizebilir	Akıllı yönetim sağlar

job çalışma şekline gelirsek;

job oluşturulur ve bir yada birden fazla adım eklenir. her adımda bir komut ve prosedür çalıştırılır. job'a schedule (zamanlama) atanır. sql server agent da bu job'u takip eder ve zamanında çalışmasını sağlar.

Senaryo	Nasıl bir Job yapılır?
Her gece veritabanı yedeği almak	Backup job
Her sabah saat 5'te rapor üretmek	Raporlama job
Eski log dosyalarını her ay temizlemek	Temizlik job
Veritabanı bakımı yapmak (index rebuild)	Bakım job'ı

örnekler:

A:

Her Gece Veritabanı Yedeği Alma (Backup Job)

Amaç: Her gece saat 02:00'de veritabanını yedekle

```
BACKUP DATABASE OgresciDB
TO DISK = 'C:\Yedekler\OgresciDB_yedek.bak'
WITH INIT, STATS = 10;
---
--burada job her gece saat 2'de yedeklenir ve veritabanının yedeğini alır.
```

B:

Haftalık İndex Rebuild Yapma (Bakım Job'u)

amacı ise her pazar gecesi tabloların indexlerinin yeniden oluşmasını sağlamak.

```
ALTER INDEX ALL ON Ogresciler
REBUILD;
```

Hangi gün çalışacağını (Pazar mı, Pazartesi mi?) belirleyen şey SQL kodu değil. Bunu Job'ın SCHEDULE ayarıyla yapman gerekiyor.

C:

Eski Kayıtları Silme (Temizlik Job'u)

Amaç: 1 yıldan eski log kayıtlarını sil

```
DELETE FROM Loglar
WHERE Tarih < DATEADD(YEAR, -1, GETDATE());
```

D:

günlük rapor oluşturma

```
INSERT INTO Rapor_AktifOgresciler (Ad, Soyad, Bolum)
SELECT Ad, Soyad, Bolum
FROM Ogresciler
WHERE Durum = 'Aktif';
```

```
--ilk başta "Rapor_AktifOgrenciler" tablosu oluşturuluydu
--bu yüzden oluşturdum ki hata vermesin
```

```
CREATE TABLE Rapor_AktifOgrenciler (
    Ad VARCHAR(50),
    Soyad VARCHAR(50),
    Bolum VARCHAR(50)
);
```

SCHUDULE

Bir Job'ın ne zaman çalışacağını belirler. Yani **Job'ı tetikleyen zaman planıdır**.

Yöntem	Açıklama
GUI (grafik arayüzü ile)	SQL Server Management Studio'da sağ tıklayarak kolayca ayarlarsın
Kodla (T-SQL komutlarıyla)	Daha gelişmiş işler için T-SQL kullanırsın

bunu sql kodu ile yapmak istersem

```
-- 1. Schedule oluştur
USE msdb;
GO
EXEC sp_add_schedule
    @schedule_name = N'PazarGecesiSchedule',
    @freq_type = 8, -- Weekly
    @freq_interval = 1, -- Sunday
    @active_start_time = 020000; -- Saat 02:00:00

-- 2. Mevcut bir Job'a Schedule bağla
EXEC sp_attach_schedule
    @job_name = N'OgrenciIndexJob',
    @schedule_name = N'PazarGecesiSchedule';

/*
@freq_type = 8 Weekly (haftalık)
@freq_interval = 1 Pazar günü
@active_start_time = 020000 02:00 sabah çalışacak
*/
```

örnekler:

A:

job oluşturmayla başlıyoruz ve bu komut yeni bir job oluşturur, job'u aktif eder (enabled=1) ve açıklmasını verir.

```
USE msdb;
GO

EXEC sp_add_job
    @job_name = N'OgrenciIndexRebuildJob',
    @enabled = 1,
    @description = N'Ogrenciler tablosunun indexlerini her
                    Pazar gece 2:00 de rebuild eder.';

EXEC sp_add_jobstep
    @job_name = N'OgrenciIndexRebuildJob',
    @step_name = N'Indexleri Rebuild Et',
    @subsystem = N'TSQL',
    @command = N'
        ALTER INDEX ALL ON Ogrenciler
        REBUILD;
    ',
    @retry_attempts = 0,
    @retry_interval = 0,
    @on_success_action = 1; -- 1 = Next Step or Quit
/*
Bu komut şunu yapar: Job'a bir adım ekler (step)
Bu adımda ALTER INDEX komutu çalışır
*/

EXEC sp_add_schedule
    @schedule_name = N'PazarGecesiSchedule',
    @freq_type = 8,          -- Weekly
    @freq_interval = 1,      -- Sunday
    @freq_subday_type = 1,   -- At a specific time
    @active_start_time = 020000, -- 02:00 AM
    @active_start_date = 20240430; -- Bugünden itibaren aktif
/*
Bu komut şunu yapar:
Yeni bir Schedule (zamanlama) oluşturur
Haftalık, Pazar günü, saat 02:00'de tetikler
*/

EXEC sp_attach_schedule
    @job_name = N'OgrenciIndexRebuildJob',
    @schedule_name = N'PazarGecesiSchedule';
/* job schedule'u birleştiriyoruz bu komutta da job ve schedule'ı
```

ilişkilendiriyoruz, artık OgrenciIndexRebuildJob → PazarGecesiSchedule zmnında çalışır.
*/

Adım	Komut	Ne yapıyor?
1	<code>sp_add_job</code>	Yeni bir Job oluşturur
2	<code>sp_add_jobstep</code>	Job'a yapılacak adımı ekler
3	<code>sp_add_schedule</code>	Zamanlama (Schedule) oluşturur
4	<code>sp_attach_schedule</code>	Job ile zamanlamayı bağlar

TRANSACTION

Transaction, bir veya daha fazla SQL komutunun **tek bir bütün olarak çalışmasını sağlayan bir yapıdır**. Amaç: **ya hepsi başarılı olsun**, ya da **hiçbiri olmasın** (yani geri alınsın).

Bir sipariş işleminde şu adımlar vardır:

1. Stoktan ürün düşülür
2. Müşteri borcu güncellenir
3. Sipariş kaydı oluşturulur

Eğer bu adımlardan biri başarısız olursa, diğerleri de iptal edilmelidir!

İşte **Transaction** bu işi yapar!

Amaç	Açıklama
Veri tutarlılığı	Tüm işlem başarıyla tamamlanmalı (ya hep ya hiç)
Hata yönetimi	Hata olduğunda tüm işlem geri alınır (ROLLBACK)
Güvenlik	Kısmi güncelleme veya silme riskini ortadan kaldırır
Çoklu işlem kontrolü	Birden fazla tabloya işlem yapılırken koordinasyon sağlar

```
BEGIN TRANSACTION;

-- 1. işlem
UPDATE Urunler SET Stok = Stok - 5 WHERE UrunID = 1;

-- 2. işlem
INSERT INTO Siparisler (UrunID, Adet) VALUES (1, 5);

-- 3. işlem (hata olabilir!)
-- Mesela yanlış tablo adı varsa:

IF @ERROR <> 0
    ROLLBACK;
ELSE
```

```
COMMIT;
```

```
-- commit işlemi kalıcı hale getirir.  
-- rollback her şeyi geri alır.
```

transaction türlerine gelirsek de

1. Explicit Transaction (Açık Transaction)

Sen kendin **BEGIN TRANSACTION** yazarak başlatırsın

```
BEGIN TRANSACTION  
-- işlemler  
COMMIT / ROLLBACK
```

2. Implicit Transaction (Kapalı Transaction)

SQL Server ayarlandığında her komutu **otomatik** olarak bir transaction olarak çalıştırır. Sen yazmazsın ama arkada çalışır.

```
SET IMPLICIT_TRANSACTIONS ON;
```

3. Autocommit Mode (Varsayılan)

Her komut otomatik commit olur. Transaction yazmadan işlem anında kalıcı hale gelir.

Ama geri alma şansın olmaz!

Transaction'ların en önemli özellikleri 4 temel ilkeye dayanır:

Kural	Anlamı
Atomicity	Ya hep ya hiç
Consistency	Veri tutarlılığı korunur
Isolation	İşlemler birbirini etkilemez
Durability	Commit edilmiş işlem kalıcıdır

Durum	Transaction Gerekli mi?
Tek bir tabloya SELECT	✗ Gerekmez
Birden fazla tabloya INSERT/UPDATE	✓ Gerekli
Finansal işlemler (banka vb.)	✓ Kesinlikle gerekir
Hataya karşı işlem güvenliği	✓ Olmazsa olmaz

```
BEGIN TRANSACTION;  
BEGIN TRY  
    UPDATE Hesaplar SET Bakiye = Bakiye - 100 WHERE ID = 1;  
    UPDATE Hesaplar SET Bakiye = Bakiye + 100 WHERE ID = 2;  
    COMMIT;  
END TRY  
BEGIN CATCH  
    ROLLBACK;  
    PRINT 'İşlem başarısız, geri alındı.';  
END CATCH;
```

```
/*
Transaction, işlemleri bir bütün haline getirir
Hata olursa geri alınır
BEGIN TRANSACTION, COMMIT, ROLLBACK komutları kullanılır
En çok: finans, stok, sipariş, muhasebe işlemlerinde kullanılır
*/
```

örnekler:

1.örnek başarılı sipariş işlemi

Senaryo:

Müşteri sipariş veriyor.

→ Stok düşülecek

→ Sipariş kaydı girilecek

Eğer **her iki işlem de başarılı olursa** commit edilir. Aksi durumda geri alınır.

```
BEGIN TRANSACTION;
BEGIN TRY
    -- 1. stok düş
    UPDATE Urunler
    SET Stok = Stok - 2
    WHERE UrunID = 1;

    -- 2. siparişi kaydet
    INSERT INTO Siparisler (UrunID, Adet, Tarih)
    VALUES (1, 2, GETDATE());

    COMMIT; -- her şey başarılıysa
    PRINT 'Sipariş başarıyla oluşturuldu!';
END TRY
BEGIN CATCH
    ROLLBACK; -- bir yerde hata varsa geri al
    PRINT 'Sipariş sırasında hata oluştu, işlemler geri alındı!';
END CATCH;
```

ÖRNEK 2 – Maaş Güncelleme: Hatalıysa iptal et

Senaryo: Bir çalışanın maaşı güncelleniyor. Ama güncellenen maaş **10.000'den büyükse** bu işlem iptal edilsin.

```
BEGIN TRANSACTION;
DECLARE @YeniMaas MONEY = 12000;
IF @YeniMaas > 10000
BEGIN
    PRINT 'Maaş çok yüksek, işlem iptal edildi.';
```

```

ROLLBACK;
END
ELSE
BEGIN
    UPDATE Calisanlar
    SET Maas = @YeniMaas
    WHERE CalisanID = 5;
    COMMIT;
    PRINT 'Maaş güncellendi.';
END;

```

ÖRNEK 3 –

Hesaplar arası para transferi

Senaryo: Kullanıcı A'dan para çekip, kullanıcı B'ye aktarıyoruz. Bu işlemin ortasında hata olursa her şey geri alınmalı.

```

BEGIN TRANSACTION;
BEGIN TRY
    -- Hesap 1'den para çek
    UPDATE Hesaplar
    SET Bakiye = Bakiye - 500
    WHERE HesapID = 1;
    -- Hesap 2'ye para yatır
    UPDATE Hesaplar
    SET Bakiye = Bakiye + 500
    WHERE HesapID = 2;
    COMMIT;
    PRINT 'Para başarıyla transfer edildi.';
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Transfer sırasında hata oluştu, işlemler geri alındı.';
END CATCH;

```

- Tüm örneklerde **ya hep ya hiç** kuralı geçerli
- **BEGIN TRANSACTION** başlatır, **COMMIT** kalıcı yapar, **ROLLBACK** geri alır
- TRY...CATCH bloğu sayesinde hata olduğunda rollback otomatik yapılabilir

HATALI OLAN KODLAR;

SORU 1 – Hatalı ROLLBACK Kullanımı


```

BEGIN TRANSACTION;

UPDATE Urunler
SET Stok = Stok - 3
WHERE UrunID = 1;

IF @ERROR <> 0
    PRINT 'Hata oluştu!';
    ROLLBACK;
COMMIT;
-- burada roll back ifadesi yanlış kullanılmış.

```

soru2:

```

BEGIN TRANSACTION;
UPDATE Hesaplar
SET Bakiye = Bakiye - 100
WHERE HesapID = 1;
UPDATE Hesaplar
SET Bakiye = Bakiye + 100
WHERE HesapID = 2;

IF @@ERROR <> 0
    ROLLBACK;
ELSE
    COMMIT;

/*
@@ERROR sadece en son yapılan işlem için geçerlidir.
→ İlk UPDATE başarılı olabilir ama ikinci UPDATE başarısızsa, sadece onu kontrol eder.
Ama ya ilk UPDATE hata veriyse?
→ O zaman @@ERROR sıfırlanmış olur ve sen hatayı göremezsin! 😞
TRY...CATCH bloğu yok.
→ Hata olursa SQL Server işlemi durdurur, ROLLBACK çalışmaz bile.
*/

-- DOGRUSU İSE

BEGIN TRY
    BEGIN TRANSACTION;
    UPDATE Hesaplar
    SET Bakiye = Bakiye - 100
    WHERE HesapID = 1;
    UPDATE Hesaplar
    SET Bakiye = Bakiye + 100

```

```

WHERE HesapID = 2;
COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Hata oluştu, işlemler geri alındı!';
END CATCH;
/*
TRY...CATCH → tüm işlemleri sarar
Hangi UPDATE hatalı olursa olsun CATCH'e düşer
ROLLBACK her şeyi geri alır
*/

```

soru3:

```

BEGIN TRANSACTION;
BEGIN TRY
    DELETE FROM Siparisler WHERE SiparisID = 999;
    PRINT 'Sipariş silindi!';
END TRY
BEGIN CATCH
    PRINT 'Hata oluştu!';
    ROLLBACK;
END CATCH;

/*
HATA NEREDE?
**COMMIT yok!**
Yani işlem başarılı da olsa **kalıcı olmuyor.**
SQL Server bu işlemi **beklemede (pending)** tutar.
Bağlantı kapandığında işlem otomatik olarak geri alınabilir.
*/
BEGIN TRANSACTION;

BEGIN TRY
    DELETE FROM Siparisler WHERE SiparisID = 999;

    COMMIT;
    PRINT 'Sipariş silindi!';
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Hata oluştu, geri alındı!';
END CATCH;

/*

```

NEDEN ÖNEMLİ?

BEGIN TRANSACTION'la başlıyorsan, ya COMMIT ya da ROLLBACK ile bitirmelisin.
Aksi halde işlem yarım kalır → veri görünürde silinmiş gibi olur ama aslında silinmez.
*/

Durum	Açıklama
COMMIT unutulursa	İşlem başarılı olsa bile kalıcı olmaz
ROLLBACK yoksa	Hata anında veri geri alınmaz
@@ERROR sadece 1 işlem için	Tüm işlem bloğu için işe yaramaz
TRY...CATCH	En güvenli yöntemdir

MDF VE LDF DOSYALAR

Dosya Türü	Açılımı	Görevi
MDF	Primary Data File	Veritabanının asıl verilerini tutar
LDF	Log Data File	Veritabanındaki işlem günlüklerini tutar

1. MDF – Main Data File (Ana Veri Dosyası)

Görevi:

- Tablo yapıları
- Satır verileri
- Index'ler
- View'lar, Stored Procedure'lar Her şey MDF dosyasında tutulur.

C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\Okul.mdf

2. LDF – Log Data File (Kayıt/Günlük Dosyası)

Görevi:

- Her SQL komutu çalıştığında **loglanır**
- Bu loglar geri alma (ROLLBACK), kurtarma, yedekleme işlemlerinde kullanılır
- Yani "**ne yapıldı, ne zaman yapıldı, kimin tarafından yapıldı**" gibi bilgiler burada

C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\Okul_log.ldf

NEDEN İKİ AYRI DOSYA VAR?

Çünkü veriyle logların ayrı tutulması: Veritabanını daha **güvenli** yapar, **Performans** sağlar (bir dosyada okuma, diğerinde yazma olabilir), **Kurtarma** işlemlerini kolaylaştırır, Backup/restore sistemleri bu ayrımı kullanır.

```
CREATE DATABASE Okul;  
/*  
SQL Server arka planda otomatik olarak:  
Okul.mdf (veri dosyası)  
Okul_log.ldf (log dosyası)  
oluşturur. Bu dosyalar .mdf ve .ldf uzantısıyla diskte saklanır. */
```

NOT: Bunlar silinirse ne olur?

MDF silinirse → **Verilerin tamamı kaybolur**

LDF silinirse → Veritabanı **başlamaz**, çünkü işlem günlükleri eksiktir

Yani her iki dosya da çok kıymetlidir. Asla silinmemeli!

NDF Dosyası Nedir?


- Büyük projelerde ek veri dosyaları gerekebilir
- İşte o zaman **NDF** (Secondary Data File) kullanılır
- Ana dosya hâlâ **MDF** 'dir ama **NDF** ilave veri alanı sağlar

Kısaltma	Dosya Türü	Görevi
MDF	Primary Data File	Veritabanı verilerini saklar
LDF	Log Data File	İşlem loglarını, rollback'leri tutar
NDF	Secondary Data File	Ek veri depolama alanı sağlar (opsiyonel)

SORU 1: Kavramsal Bilgi


 Aşağıdakilerden hangisi **SQL Server veritabanı dosya yapısı** hakkında doğrudur?

- A) MDF dosyası yalnızca işlem loglarını tutar.
- B) LDF dosyası, kullanıcı tabloları ve view'ları içerir.
- C) MDF dosyası, veritabanının asıl veri içeriğini saklar.
- D) LDF dosyası silinirse veriler korunur ve sistem normal çalışır.

 Doğru cevap: C


! Çünkü MDF → tablo verileri, stored procedure'lar gibi tüm veriyi tutar.
LDF → logları.

SORU 2: Uygulamalı Senaryo

 Aşağıdaki SQL komutunu çalıştırdığınızda SQL Server arka planda hangi dosyaları oluşturur?

```
sql
KopyalaDüzenle
CREATE DATABASE OkulDB;
```

- A) Sadece .mdf dosyası
- B) .mdf ve .ndf dosyaları
- C) .mdf ve .ldf dosyaları
- D) .ldf ve .bak dosyaları

 Doğru cevap: C

Çünkü veritabanı oluşturulunca:


OkulDB.mdf → ana veri dosyası,

OkulDB_log.ldf → log dosyası otomatik oluşur.

SORU 3: Hata Yönetimi

 Aşağıdaki durumların hangisinde veritabanı **başlatılamaz**?

- A) .bak yedeği eksikse
- B) .ndf dosyası silinmişse ama kullanılmıyorsa
- C) .ldf dosyası silinmişse
- D) Veritabanı adı değiştirilmişse

 Doğru cevap: C

Çünkü .ldf silinirse SQL Server veritabanını **başlatamaz**. Transaction log dosyası yoksa veri tutarlılığı garanti edilemez!

AÇIK UÇLU SORULAR;

Soru 1:

Bir SQL Server veritabanı oluşturduğunda MDF ve LDF dosyaları neden ayrı tutulur? Bunun veri güvenliği ve performans açısından avantajlarını açıkla.

Cevap:

MDF dosyası veritabanının ana veri dosyasıdır; tablolar, veriler, view'lar gibi yapıları içerir.

LDF ise transaction log (işlem günlükleri) dosyasıdır. Her işlem (INSERT, DELETE, UPDATE) LDF'de kayıt altına alınır.

Ayrı tutulmasının avantajları:

- **Veri güvenliği:** Olası sistem çökmesinde LDF sayesinde işlemler geri alınabilir veya yeniden yapılabilir.
- **Performans:** Veri okuma işlemleri MDF'den, yazma işlemleri LDF'ye yapıldığı için paralel çalışabilir, bu da hızı artırır.
- **Yedekleme:** Log'lar ayrı alındığı için farklı zamanlarda alınabilir.

Soru 2:

LDF dosyasının sistem için önemi nedir? Silinirse ne olur, SQL Server bu durumda nasıl davranır?

Cevap:

LDF dosyası, veritabanındaki işlemlerin kaydını tutar. Her işlem (örneğin, bir satır silinmesi) önce LDF'ye yazılır.

Silinirse:

- SQL Server **veritabanını başlatamaz**
- "Recovery Pending" hatası alırsın
- Çünkü SQL Server, veri bütünlüğünü koruyamaz

Kurtarma için:

- Log dosyasını yeniden oluşturman gerekebilir (`DBCC CHECKDB` ya da `ATTACH_REBUILD_LOG`)
- Ya da yedekten veritabanını geri yüklemen gerekir

Soru 3:

Bir veritabanı yedeği `.bak` dosyasından geri yüklendiğinde, MDF ve LDF dosyaları da otomatik oluşur mu? Oluşmuyorsa ne yapılmalıdır?

Cevap:

Evet, genellikle `.bak` yedeğinden geri yükleme yapılırken SQL Server **MDF ve LDF dosyalarını otomatik oluşturur.**

Ancak istersen dosya konumlarını değiştirebilirsin:

```
sql
KopyalaDüzenle
RESTORE DATABASE Okul
FROM DISK = 'C:\Yedekler\Okul.bak'
WITH MOVE 'Okul_Data' TO 'D:\Veriler\Okul.mdf',
     MOVE 'Okul_Log' TO 'D:\Loglar\Okul.ldf';
```

Eğer yedek eksikse veya dosyalar bozulmuşsa, SQL Server yüklemeyi reddeder.

Soru 4:

NDF dosyasının kullanım amacı nedir? Büyük bir projede neden ek `.ndf` dosyalarına ihtiyaç duyulabilir?

Cevap:

NDF dosyası, MDF dosyasının büyüklüğü yetmediğinde veya performansı artırmak için kullanılan **ek veri dosyasıdır**.

Veriler MDF ve NDF dosyalarına bölünebilir.

Kullanım nedenleri:

- Disk alanı yetersizse başka diske veri taşıma imkanı sağlar
- Büyük veritabanlarında **veri yükünü dağıtarak** performansı artırır
- Dosya grupları ile veri yönetimini kolaylaştırır

Soru 5:

MDF dosyasının bozulduğu ya da erişilemez hale geldiği bir senaryoda veri kurtarımı yapılabilir mi? SQL Server'ın hangi özellikleri bu duruma çözüm olabilir?

Cevap:

Eğer MDF bozulursa, veriler doğrudan kaybedilebilir. Ancak bazı kurtarma yolları vardır:

- **.bak** yedeği varsa **RESTORE** ile geri yükleme yapılabilir
- Eğer sadece MDF bozuksa ve LDF sağlamısa, **log'dan veri kurtarma** denenebilir
- **DBCC CHECKDB** komutu ile onarma işlemi yapılabilir
- **EMERGENCY MODE** ile sadece okuma yapılabilir

Ancak bu tür durumlar için düzenli yedek alma **hayati önem taşır**.

Soru 6:

Veritabanı yöneticisi (DBA) olarak bir veritabanı oluşturdu. Bu veritabanının veri dosyalarının konumunu, boyutlarını ve büyüme ayarlarını manuel olarak nasıl belirlersin?

Cevap:

```
sql
KopyalaDüzenle
CREATE DATABASE Okul
ON
(
    NAME = Okul_Data,
    FILENAME = 'D:\Veri\Okul.mdf',
    SIZE = 100MB,
    MAXSIZE = 2GB,
    FILEGROWTH = 10MB
)
LOG ON
(
    NAME = Okul_Log,
    FILENAME = 'D:\Loglar\Okul_log.ldf',
    SIZE = 50MB,
```

```
MAXSIZE = 1GB,  
FILEGROWTH = 5MB  
);
```

Bu şekilde:

- Dosya yollarını belirlemiş olursun
- Sabit boyut ve büyüme oranları ile **disk yönetimini kontrol altında tutarsın**

TRIGGER

triggerlar bir tablo üzerinde insert, update veya delete işlemi gerçekleştiğinde otomatik olarak çalışan bir sql komut bloğudur adı üstünde tetikleyici gibi davranır.

ne işe yaradığına gelirsek de şöyle ki;

değişiklik olduğunda otomatik tepki verir veri güvenliğini artırır log kayıtları tutabilir, kuralları zorlayabilir, başka tabloları etkileyebilir arkaplanda işlem yapabilir(örneğin stok güncellemesi).

Tür	Açıklama
AFTER Trigger (FOR)	INSERT , UPDATE , DELETE işleminden sonra çalışır
INSTEAD OF Trigger	Normal işlemi engelleyip, kendi işlemini yapar
DDL Trigger	CREATE , DROP , ALTER gibi tablo/değişiklik işlemleri için kullanılır

a: after insert örneği

```
CREATE TRIGGER tr_OgrenciEklendiginde  
ON Ogrenciler  
AFTER INSERT  
AS  
BEGIN  
    PRINT 'Yeni öğrenci eklendi!';  
END;  
-- bu triggerda ogrenciler tablosuna yeni bir kayıt eklendiğinde  
--otomatik olarak çalışmasını sağlar
```

Özellik	Açıklama
Trigger	Otomatik çalışır (INSERT , DELETE , UPDATE)
AFTER Trigger	İşlemden sonra devreye girer
INSTEAD OF Trigger	İşlemi engeller, yerine kendi işlemini yapar
INSERTED / DELETED	Geçici tablolarla işlem verisini alır

Trigger çeşitlerine gelecek olursak

1. AFTER TRIGGER

Veritabanında **INSERT**, **UPDATE** veya **DELETE** işlemi **başarıyla gerçekleştikten sonra çalışır**.

En yaygın kullanılan türdür.

kayıt eklendiğinde log tutmak için, veri güncellendiğinde başka tabloyu tetiklemek için, silinen veriyi yedek tabloya taşımak için kullanılabilir.

```
CREATE TRIGGER tr_OgrenciEklendi
ON Ogrenciler
AFTER INSERT
AS
BEGIN
    PRINT 'Yeni öğrenci eklendi!';
END;
-- AFTER SADECE TABLOLARDAN KULLANILIR.
```

2. **INSTEAD OF TRIGGER**

INSERT, UPDATE, DELETE işlemini engeller ve kendi komutunu çalıştırır. Özellikle view'ler üzerinde kullanılır çünkü viewler doğrudan güncellenemezler.

Kullanıldığı yerler:

- **View** üzerine yazılıp, gerçek tablolara yönlendirme
- Güncelleme/silme kısıtlaması olan verilerde işlem kontrolü
- Kısıtlı veri güncelleme mantığı (örneğin sadece aktif kullanıcılar)

```
CREATE TRIGGER tr_VwOgrenciGuncelle
ON vw_AktifOgrenciler
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE Ogrenciler
    SET Ad = i.Ad
    FROM Ogrenciler o
    JOIN INSERTED i ON o.OgrenciID = i.OgrenciID
    WHERE o.Durum = 'Aktif';
END;
-- INSTEAD OF VIEWLER İLE ÇALIŞABİLEN NET TRIGGER TÜRÜDÜR.
```

3. **DDL TRIGGER drop işlemini engelle**

CREATE, ALTER, DROP gibi veritabanı yapısını değiştiren işlemleri tetikler, tabloya değil veritabanına ve sunucuya yazılır.

Kullanıldığı yerler:

- Tablo silinmesin diye güvenlik amaçlı
- Tablo yaratıldığında log kaydı
- Şema değişikliği izleme
- Yetkisiz kullanıcıların yapısal değişikliklerini engellemek

```
CREATE TRIGGER tr_TablosilmeEngelle
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
    PRINT 'Tablo silme işlemi engellendi!';
    ROLLBACK;
END;
```

Trigger Türü	Tetikleyen İşlem	Kullanıldığı Yer
AFTER INSERT/UPDATE/DELETE	Veri işlemi sonrası	Tablo
INSTEAD OF	Veri işlemini override eder	Genelde View'lar
DDL Trigger	Yapısal işlemler (CREATE , DROP , ALTER)	Veritabanı veya sunucu seviyesi

"AFTER trigger'lar tablo işlemleri sonrası çalışır, INSTEAD OF trigger'lar işlemi engelleyip yönlendirir, DDL trigger'lar ise veritabanı yapısal işlemlerinde devreye girer."

örnekler:

1. insert sonrası log tablosu oluşturma;

```
-- ilk önce log tablosunu oluşturalım.
CREATE TABLE OgrenciLog (
    Ad VARCHAR(50),
    IslemTarihi DATETIME,
    IslemTipi VARCHAR(20)
);

-- şimdi ise trigger oluşturalım.
CREATE TRIGGER tr_Ogrenci_Insert_Log
ON Ogrenciler
AFTER INSERT
AS
```

```
BEGIN
  INSERT INTO OgrenciLog (Ad, IslemTarihi, IslemTipi)
  SELECT Ad, GETDATE(), 'Ekleme'
  FROM INSERTED;
END;
```

Sanal Tablo	Ne içerir?
INSERTED	Yeni eklenen verileri
DELETED	Silinen veya güncellenmeden önceki verileri

!!! bu sanal tablolar sadece trigger içinde kullanılacağını unutma

2. öğrenci silmek istediğinde gerçekten silmesin sadece pasif yapsın.

```
--ilk önce öğrenciler tablosuna durum sütunu ekledim ki
--aktif yada pasif olduğu gösterebileyim
ALTER TABLE Ogrenciler
ADD Durum VARCHAR(10) DEFAULT 'Aktif';

-- şimdi ise triggerımı ekliyorum.
CREATE TRIGGER tr_OgrenciInsteadOfDelete
ON Ogrenciler
INSTEAD OF DELETE
AS
BEGIN
  UPDATE Ogrenciler
  SET Durum = 'Pasif'
  WHERE OgrenciID IN (SELECT OgrenciID FROM DELETED);
END;
```

3. Tablo silinmesin diye güvenlik kontrolü

```
CREATE TRIGGER tr_EngelleTabloSilme
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
  PRINT 'Tablo silinemez!';
  ROLLBACK;
END;

/*
biri drop table yaptığı zaman işlemi iptal eder rollback ile işlemi geri alır.
```

kullanıcıya uyarı mesajı verir.

*/

sınava yönelik sorular;

1. AFTER UPDATE – Ücret Zam Takibi

Eğer bir çalışanın maaşı güncellenirse, zam kaydını **MaasZamLog** tablosuna ekle.

```
CREATE TABLE Calisanlar (  
    ID INT PRIMARY KEY,  
    Ad VARCHAR(50),  
    Maas DECIMAL(10,2)  
);  
  
CREATE TABLE MaasZamLog (  
    ID INT IDENTITY PRIMARY KEY,  
    CalisanID INT,  
    EskiMaas DECIMAL(10,2),  
    YeniMaas DECIMAL(10,2),  
    Tarih DATETIME  
);  
  
CREATE TRIGGER tr_MaasGuncelle  
ON Calisanlar  
AFTER UPDATE  
AS  
BEGIN  
    INSERT INTO MaasZamLog (CalisanID, EskiMaas, YeniMaas, Tarih)  
    SELECT i.ID, d.Maas, i.Maas, GETDATE()  
    FROM INSERTED i  
    JOIN DELETED d ON i.ID = d.ID  
    WHERE i.Maas <> d.Maas;  
END;  
  
/*  
Neden kullanılır?  
Kim ne zaman ne kadar zam aldı izlemek için  
Geriye dönük kontrol yapabilmek için  
İnsan hatasını azaltmak için  
  
Nasıl çalışır?  
UPDATE işlemi başarıyla tamamlanınca çalışır  
DELETED tablosu eski maaşı getirir  
INSERTED tablosu yeni maaşı getirir
```

İkisi karşılaştırılır, farklıysa log atılır.

*/

2. INSTEAD OF INSERT – Negatif miktar engelleme

Bir ürüne negatif stok girişi yapılmasını engelle.

```
CREATE TABLE UrunStok (  
    UrunID INT PRIMARY KEY,  
    UrunAdi VARCHAR(50),  
    Miktar INT  
);  
CREATE TRIGGER tr_StokKontrol  
ON UrunStok  
INSTEAD OF INSERT  
AS  
BEGIN  
    IF EXISTS (SELECT * FROM INSERTED WHERE Miktar < 0)  
    BEGIN  
        RAISERROR('Negatif stok girişi yapılamaz!', 16, 1);  
        ROLLBACK;  
    END  
    ELSE  
    BEGIN  
        INSERT INTO UrunStok (UrunID, UrunAdi, Miktar)  
        SELECT UrunID, UrunAdi, Miktar FROM INSERTED;  
    END  
END;
```

/*

Ne yapıyor?

Birisi UrunStok tablosuna negatif stok miktarıyla kayıt eklemeye kalkarsa işlemi durduruyor ve hata veriyor.

Neden kullanılır?

Veri bütünlüğünü korumak için

Stok değerlerinin negatif olmasını engellemek için

Hatalı kullanıcı girişlerine karşı otomatik koruma sağlar

Nasıl çalışır?

INSERT işlemini engelliyor

INSERTED tablosunda negatif varsa ROLLBACK yapıyor

Yoksa INSERT işlemini manuel olarak yapıyor

Bu trigger sayesinde veritabanına yanlış veri girişi yapılamıyor.*/

3. DDL TRIGGER – DROP INDEX engelleme

Kimse veritabanında index silmesin.

```
CREATE TRIGGER tr_IndexSilmeEngelle
ON DATABASE
FOR DROP_INDEX
AS
BEGIN
    PRINT 'Index silinemez!';
    ROLLBACK;
END;
```

/*

Ne yapıyor?

Herhangi bir kullanıcı, veritabanındaki bir index'i silmeye kalkarsa, otomatik olarak işlemi durduruyor.

Neden kullanılır?

Yetkisiz kişilerin veri yapısını bozmasını engellemek için

Bilerek ya da kazara yapılan silme işlemlerine karşı koruma sağlamak için

Gelişmiş güvenlik önlemidir

Nasıl çalışır?

SQL Server'da biri DROP INDEX komutu verirse

Trigger devreye giriyor ve ROLLBACK yaparak işlemi iptal ediyor

Kullanıcıya "index silinemez" mesajı gösteriliyor

Bu tür trigger'lar genelde DBA (veritabanı yöneticisi) tarafından kullanılır.

*/

4. AFTER DELETE – Sipariş Silinince Logla

Sipariş silindiğinde bilgileri **SilinenSiparisler** tablosuna kaydet.

```
CREATE TABLE Siparisler (
    SiparisID INT,
    Urun VARCHAR(50),
    Adet INT
);
```

```
CREATE TABLE SilinenSiparisler (
    SiparisID INT,
    Urun VARCHAR(50),
```

```

Adet INT,
SilinmeTarihi DATETIME
);

CREATE TRIGGER tr_SiparisSilindi
ON Siparisler
AFTER DELETE
AS
BEGIN
    INSERT INTO SilinenSiparisler (SiparisID, Urun, Adet, SilinmeTarihi)
    SELECT SiparisID, Urun, Adet, GETDATE()
    FROM DELETED;
END;
/*
Ne yapıyor?
Siparişler tablosundan bir kayıt silindiğinde, silinen satır
SilinenSiparisler adlı başka bir tabloya kaydediliyor.

Neden kullanılır?
Silinen verileri kaybetmemek için
Kullanıcı "yanlışlıkla sildim" dediğinde geri getirmek için
Raporlama ve denetim için

Nasıl çalışır?
DELETE işlemi başarıyla tamamlandığında devreye girer
DELETED sanal tablosu silinen veriyi içerir
Bu veri aynen başka tabloya yazılır (GETDATE() ile silinme zamanı da eklenir)
*/

```

Tür	Ne Yapar	Kullanıldığı Yer
AFTER UPDATE	Maaş değişimi gibi olayları loglar	Güncelleme sonrası
INSTEAD OF INSERT	Kurallı veri eklemeyi zorunlu kılar	INSERT işlemi engellenip kontrol yapılır
DDL TRIGGER	Yapısal komutları engeller	DROP INDEX, DROP TABLE
AFTER DELETE	Silinen veriyi başka tabloya taşır	Kayıt geçmişi tutmak için

Siparisler tablosuna yeni sipariş eklenince, **Stok** tablosundaki ürün miktarını 1 azaltan bir trigger yazınız.

HATA YÖNETİMİ VE KULLANIM ÖRNEKLERİ BAKIM PLANLAMA

HATA TÜRLERİ;

SYNTAX - SÖZDİZİMİ HATALARI;

sql kodunun yanlış yazılması olayına deniyor.

RUNTIME HATALARI- ÇALIŞMA ZAMANI;

sorgu veya işlem çalıştırılırken ortaya çıkan hatalara deniyor. sıfıra bölme, veri tipi dönüşümü, sayısal taşma, nesne bulunamaması gibi örnekler verilebilir.

CONSTRAINT - KISITLAMA

- **Primary Key İhlali:** Primary Key sütununa zaten var olan bir değeri veya NULL(eğer izin verilmiyorsa) eklemeye çalışmak.
- **Foreign Key İhlali:** ilişkili tabloda olmayan bir değere referans verme.
- **Unique Constraint İhlali:** UNIQUE olması gereken bir alana tekrar eden bir değer girme.
- **Check Constraint İhlali:** Tanımlı kurala (örn. Price > 0) uymayan veri girme.
- **NULL Kısıtlaması İhlali:** NOT NULL olarak girilmiş bir sütuna NULL değer eklemeye çalışmak.

PERMISSION - YETKİ

Yetki (Permission) Hataları

•Nedir?

- Kullanıcının veya uygulamanın, gerçekleştirmek istediği işlem için gerekli izne sahip olmaması.
- Genellikle "Permission denied" (İzin reddedildi) veya benzeri bir hata mesajı alırsınız.

•Örnekler:

- Bir tabloyu SELECT, INSERT, UPDATE, DELETE etme yetkisinin olmaması.
- Bir Stored Procedure'ü çalıştırma yetkisinin olmaması.
- Bir veritabanına veya şemaya erişim yetkisinin olmaması.

•Nasıl Giderilir?

- Hangi kullanıcı(login) ile veritabanına bağlı olduğu kontrol edilir.
- Hata mesajını dikkatlice okunur - genellikle hangi nesne üzerinde yetki hatası olduğunu belirtir.
- Gerekli yetkilerin ilgili kullanıcıya/role tanımlanmasını sağlama.

RESOURCE- KAYNAK

•Nedir?

- SQL Server'in çalışması için gerekli sistem kaynaklarının (CPU, bellek, disk alanı, ağ bant genişliği vb.) yetersizliğinden kaynaklanan hatalardır.
- Doğrudan bir hata mesajı yerine, yavaşlama, takılma veya belirli işlemlerin başarısız olması şeklinde görülebilir.

•Örnekler:

- **Deadlock (Kilitlenme):** İki veya daha fazla işlemin birbirinin kilitlediği kaynağı beklemesi.
- **Timeout (Zaman Aşımı):** Bir sorgunun/işlemin belirlenen sürede tamamlanamaması.
- **Yetersiz Bellek:** Sorguların veya işlemlerin çalışması için yeterli RAM olmaması.
- **Disk Alanı Sorunları:** Veritabanı dosyalarının büyümesi için yeterli disk alanı olmaması.
- **Bağlantı Havuzu Sorunları:** Uygulamanın veritabanı bağlantı havuzunda sorun yaşaması.

•Nasıl Giderilir?

- Sunucu performansını izleme (CPU, RAM, Disk I/O).
- Sorgu optimizasyonu yapma.
- Deadlock'ları tespit ve giderme.
- Kaynakları artırma veya yapılandırmayı optimize etme.

DATA İNTEGRİTY VERİ BÜTÜNLÜĞÜ

•Nedir?

- Verinin tutarlılığı, doğruluğu veya güvenilirliği ile ilgili sorunlar.
- Constraint hatalarını da kapsar, ancak daha geniş anlamda.
- Genellikle daha ciddi sorunlara işaret eder ve veri kaybı riski taşır.

•Nasıl Giderilir?

- SQL Server hata logları kontrol edilir - genellikle 89xx serisi hata mesajları görülür.
- Bozulma tespit edilirse, bilinen en son sağlam yedekten geri yükleme yapılır.
- Ciddi bozulma durumlarında Microsoft Desteği ile iletişime geçmeniz gerekebilir.

•Örnekler:

- Donanım arızaları (özellikle disk sistemleri).
- İşletim sistemi veya SQL Server yazılımındaki hatalar.
- Ani güç kesintileri veya sistem çökmeleri.
- Yanlış veya eksik yedekleme/geri yükleme işlemleri.
- Bellek (RAM) sorunları.

CONNECTION BAĞLANTI

uygulamanın veya kullanıcının veri tabanına bağlanırken yaşadığı sorunlara deniyor.

•Örnekler:

- Sunucunun çalışmıyor olması.
- Ağ bağlantısı sorunları.
- Yanlış sunucu adı, IP adresi, port.
- Yanlış kullanıcı adı veya şifre.
- Firewall engelleri.

Bir SQL Server hata mesajı genellikle şu bileşenleri içerir:

1.Hata Numarası (Error Number): Her hatanın benzersiz bir numarası vardır (örneğin, 102, 547, 18456). Bu numara, hatanın türünü hızlıca tanımlamaya yardımcı olur. sys.messages sistem kataloğu görünümünde bu numaralar ve ilgili mesaj metinleri bulunur.

2.Hata Ciddiyeti (Severity Level): Hatanın ne kadar ciddi olduğunu belirten bir sayıdır (0 ile 25 arasında). Ciddiyet seviyesi, hatanın etkisini ve alınması gereken önlemleri gösterir.

3.Hata Durumu (State): Aynı hata numarasına sahip farklı hata durumlarını ayırt etmek için kullanılır. Genellikle dahili kullanım içindir.

4.Hata Mesajı Metni (Message Text): Hatanın ne olduğunu ve neden oluştuğunu açıklayan metin. Bu metin genellikle hatanın nedenini anlamak için en faydalı kısımdır.

--GÖRMEK İÇİN

```
SELECT message_id, language_id, severity, text
FROM sys.messages
WHERE language_id = 1033;
```

Hata Ciddiyet Seviyeleri (Severity Levels)

0–10: Bilgilendirme

11–16: Kullanıcı hataları

17–19: Kaynak hataları

20–25: Sistem hataları

♦ Örnek Hata Mesajı:

sql

SELECT 1 / 0;

mathematica

Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.

◆ Yaygın Hata Numaraları

- 102: Syntax hatası
- 208: Geçersiz nesne adı
- 547: FOREIGN KEY çakışması
- 2627: PRIMARY KEY çakışması
- 18456: Login hatası
- 8152: Veri kesilecek kadar uzun (truncate)

◆ sp_addmessage ile Kalıcı Mesajlar

```
sql
EXEC sp_addmessage
    @msgnum = 50001,
    @severity = 16,
    @msgtext = N'Bu özel hata sistemde tanımlandı.';
```

- 50000 üzeri numaralar özel hata mesajları içindir.

◆ Hata Mesajını Sistemden Silme

```
sql
EXEC sp_dropmessage @msgnum = 50001;
```

- Kalıcı hata mesajlarını silmek için kullanılır.

◆ RAISERROR ile Anlık Hata Fırlatma

```
sql
RAISERROR ('Bu bir özel hatadır!', 16, 1);
```

- 16: Ciddiyet seviyesi (11–16 kullanıcı hataları için)
- 1: Durum numarası (0–255 arası)

◆ RAISERROR ile Tanımlı Mesajı Fırlatma

```
sql
RAISERROR(50001, 16, 1);
```

- Tanımlanan özel hata sistem mesajı gibi çalışır.
- Kolayca tekrar kullanılabilir.

◆ TRY...CATCH ile Hata Yakalama

```
sql
BEGIN TRY
    RAISERROR('Deneme hatası', 16, 1);
END TRY
BEGIN CATCH
    PRINT ERROR_MESSAGE();
END CATCH;
```

```
IF @Fiyat < 0
    RAISERROR('Fiyat negatif olamaz!', 16, 1);
```

T-SQL'de özelleştirilmiş hata yönetimi mümkündür.

RAISERROR ve sp_addmessage birlikte kullanılarak profesyonel hata yapıları oluşturulabilir.

TRY-CATCH YAPISI:

```
BEGIN TRY
    -- Hata oluşturabilecek komut
    DELETE FROM Ogrenciler WHERE OgrenciID = 5;
END TRY
BEGIN CATCH
    PRINT 'Bir hata oluştu!';
```

```
PRINT ERROR_MESSAGE(); -- hatanın ne olduğunu gösterir
END CATCH;
```

Fonksiyon	Açıklama
<code>ERROR_MESSAGE()</code>	Hatanın açıklamasını döner
<code>ERROR_LINE()</code>	Hatanın olduğu satır numarası
<code>ERROR_NUMBER()</code>	Hata numarası
<code>ERROR_SEVERITY()</code>	Hatanın ciddiyet seviyesi (1-25)

--Kullanım Senaryosu:

```
BEGIN TRY
```

```
    BEGIN TRANSACTION;
```

```
    -- işlem 1
```

```
    UPDATE Urunler SET Stok = Stok - 10 WHERE UrunID = 1;
```

```
    -- işlem 2
```

```
    INSERT INTO Siparisler (UrunID, Adet) VALUES (1, 10);
```

```
    COMMIT;
```

```
END TRY
```

```
BEGIN CATCH
```

```
    ROLLBACK;
```

```
    PRINT 'Hata oluştu. İşlem geri alındı.';
```

```
    PRINT ERROR_MESSAGE();
```

```
END CATCH;
```

--Bu örnek, hem transaction kontrolü hem de hata yönetimi açısından sınavda %100 çıkar!



BAKIM PLANLAMA (MAINTENANCE PLAN)

Bakım planları, veritabanını otomatik olarak düzenli şekilde yedekleyen, indexleri düzenleyen, logları temizleyen görevlerdir.

Görev	Açıklama
Back Up Database	Veritabanının yedeğini alır
Check Database Integrity	Veritabanında bozulma var mı kontrol eder
Reorganize Index	Index'leri hafifçe optimize eder
Rebuild Index	Index'leri baştan oluşturur (daha derin temizlik)
Update Statistics	Sorgu optimizasyonu için istatistikleri günceller
Cleanup History	Eski backup, log kayıtlarını siler

Görev Zamanlama (Schedule):

Her bakım planı için bir **schedule (zamanlama planı)** tanımlanabilir.

Örnek:

- **Günlük saat 03:00'te** backup al
- **Her hafta pazar günü saat 02:00'de** index rebuild yap

SQL Server Agent aktif olmalı! Aksi halde bakım planları çalışmaz.

Basit Bakım Planı Örneği:

- Adım 1: Full database backup
- Adım 2: Index Rebuild
- Adım 3: Transaction log backup
- Adım 4: Temizlik (7 günden eski logları sil)

Başlık	Neden Önemli?
Hata Yönetimi	Uygulama çökmeden düzgün geri bildirim verir
Bakım Planları	Sistemin düzenli çalışmasını, veri kaybının önlenmesini sağlar
Otomasyon	Gece-gündüz sistemin kontrol altında kalmasını sağlar

SORU 1: SQL Server'da TRY-CATCH bloğu nasıl çalışır ve transaction yönetimi ile birlikte nasıl kullanılır?

Cevap:

SQL Server'da **TRY...CATCH** yapısı, bir sorguda ya da işlemler bütününde hata meydana geldiğinde kontrolü **CATCH** bloğuna geçirerek hatanın programı durdurmadan yönetilmesini

sağlar. Özellikle transaction içeren işlemlerde, hatalı adımlarda verilerin bozulmaması ve tutarlılığın korunması için kullanılır.

```
BEGIN TRY
    BEGIN TRANSACTION;

    -- işlem 1
    -- işlem 2
    -- ...

    COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;

    PRINT 'Bir hata oluştu!';
    PRINT ERROR_MESSAGE(); -- Hatanın açıklamasını verir
END CATCH;
```

--!!

Açıklama:

BEGIN TRY bloğunda hataya neden olabilecek işlemler yapılır. Her şey sorunsuz giderse COMMIT ile işlemler kalıcı hale gelir. Eğer hata olursa: CATCH bloğuna geçilir. ROLLBACK ile tüm işlemler geri alınır. Hata mesajı kullanıcıya gösterilir.

Neden kullanılır?

Hataları daha kontrollü ve güvenli biçimde ele alır. Veritabanı tutarlılığı bozulmaz. Geri alma (rollback) mekanizmasıyla veri kaybı önlenir.

--

S1: TRY-CATCH bloğu olmadan hata oluşursa SQL Server nasıl davranır?

Beklenen cevap: Sorguda hata oluşursa işlem hemen durur. Geri kalan kodlar çalışmaz. Transaction kullanılıyorsa rollback yapılmazsa veriler tutarsız kalabilir.

S2: ROLLBACK komutu neden önemlidir? TRY-CATCH ile birlikte nasıl kullanılır?

Beklenen cevap: Hata oluştuğunda yapılan işlemlerin geri alınmasını sağlar. TRY içinde işlem yapılır, CATCH içinde ROLLBACK yapılır. Verinin bütünlüğünü korur.

S3: Maintenance Plan ne işe yarar? Neden manuel yapmak yerine plan yapılır?

Beklenen cevap: Otomatik yedekleme, index düzenleme, veri bütünlüğü kontrolü gibi işlemleri zamanlanmış şekilde yapar. Manuel işlemler unutulabilir, planlar hata riskini azaltır.

S4: SQL Server Agent nedir ve bakım planlarıyla ne ilgisi vardır?

Beklenen cevap: SQL Server Agent, planlanan görevlerin arka planda çalışmasını sağlayan servis yapısıdır. Maintenance Plan'lar onun aracılığıyla belirlenen zamanlarda çalışır.



"Sistem yöneticisi olsaydın, haftalık bakım için nasıl bir plan yapardın? Adımlarıyla anlat."

- Full Backup
- Index Rebuild
- Update Statistics
- Cleanup Task
- DBCC CHECKDB

Sistem yöneticisi olsaydım, SQL Server veritabanlarının performansını ve veri güvenliğini sağlamak amacıyla haftalık bakım planı oluştururdum. Bu planı SQL Server Management Studio (SSMS) üzerinden Maintenance Plan kullanarak zamanlanmış şekilde tasarladım.

Adım 1: Full Backup (Tam Yedek Alma)

- Veritabanının tümünü haftada bir defa (örneğin Pazar günü saat 03:00'te) yedekledim.
- Olası veri kaybı durumunda geri dönüş sağlamak için bu adıma çok önem veririm.

Adım 2: Check Database Integrity

- `DBCC CHECKDB` komutuyla çalışır.
- Veritabanında bozulmuş ya da tutarsız veriler olup olmadığını kontrol ederim.

Adım 3: Rebuild Index

- Haftalık olarak tüm index'leri yeniden oluştururdum.
- Bu işlem, parçalanmaları (fragmentation) ortadan kaldırarak sorguların daha hızlı çalışmasını sağlar.

Adım 4: Update Statistics

- SQL Server'ın sorgu optimizasyonu için kullandığı istatistikleri güncelledim.
- Bu sayede sorgu planları daha akıllı hale gelir.

Adım 5: History & Backup Cleanup

- 7 günden eski yedek dosyalarını ve eski işlem loglarını silerek disk alanını temiz tutardım.
- Bu temizlik sayesinde sunucu şişmez ve sürdürülebilir olur.

Adım 6: Schedule (Zamanlama)

- SQL Server Agent kullanarak bu planı her hafta Pazar günü saat 03:00'e zamanladım.
- Gece saatlerini tercih ederim çünkü kullanıcı trafiği az olur.

Ekstra Önlemler: İşlem başarısız olursa mail bildirimi ekledim. Tüm işlemlerin loglarını tutarak gerektiğinde inceleme yapabiliydim.

Bu haftalık bakım planı sayesinde: Veritabanı düzenli optimize edilir. Performans korunur. Veri kaybı riskleri azaltılır. İş yükü otomasyona bağlanarak manuel hata riski sıfırlanır.

CUNSOR TİPLERİ

Cursor (imleç), SQL Server'da **bir sorgudan dönen verileri satır satır işlemek** için kullanılan bir yapıdır. Normalde SELECT komutları tüm sonuçları topluca işler. Ama bazen her satıra özel

işlem yapmak gerekir. İşte bu durumlarda **CURSOR devreye girer.**

NE İŞE YARAR?

- Her bir satıra özel işlem yapma imkânı verir
- Döngü (loop) gibi çalışır
- İçerisinde **IF** , **UPDATE** , **DELETE** , **PRINT** gibi komutlarla işlem yapılabilir

Durum	Açıklama
Her satıra ayrı işlem yapılacaksa	Örneğin: Her müşteriye özel mesaj gönderme
Toplu işlem yerine satır bazlı kontrol gerekiyorsa	Örneğin: Bazı veriler silinirken bazıları güncelleniyorsa
Karmaşık mantıklar, IF-ELSE gibi yapılarla veri işlenecekse	Cursor satır-satır kontrol sağlar
veriler üzerinden adım adım işlem yapılmak isteniyorsa	
satır bazlı işlem yaparak	örneğin bir tabloda dönen verilerle başka bir tabloya işlem yapılacaksa

cursor dediğimiz şey veritabanı ve uygulama arasında bir köprü görevi görerek sorgu sonucundaki her bir satıra tek tek erişilmesini ve bu satırlar üzerinde hassas şekilde işlem yapılmasını sağlar.

- 1 Satır bazında kontrol sağlar.
- 2 Her bir kayıt üzerinde farklı işlem yapılmasına olanak tanır.
- 3 SQL sorgularının dışında kalan daha esnek mantık işlemlerini uygulamaya açar.
- 4 PL/SQL veya T-SQL gibi prosedürel dillerle uyumlu çalışır.

```
DECLARE @ad VARCHAR(50);
DECLARE cur CURSOR FOR
SELECT Ad FROM Ogrenciler;
OPEN cur;
FETCH NEXT FROM cur INTO @ad;
WHILE @FETCH_STATUS = 0
BEGIN
    PRINT 'Öğrenci adı: ' + @ad;
    FETCH NEXT FROM cur INTO @ad;
END
CLOSE cur;
DEALLOCATE cur;
```

Cursor **performans açısından yavaştır**, bu yüzden çok büyük verilerde kullanılmaz. Yine de satır bazlı işlem gerekiyorsa **tek seçenek cursor'dur.**

özel mail göndermek, her bir öğrencinin notunu güncellemek, her satır için özel işlem yapmak gibi örnekler de verilebilir.

Cursor Kullanım Adımları:

1. **DECLARE CURSOR** – İmleci tanımla
2. **OPEN** – İmleci başlat
3. **FETCH NEXT** – Bir sonraki satırı al
4. **WHILE @@FETCH_STATUS = 0** – Satır olduğu sürece dön
5. **CLOSE** – Kapat
6. **DEALLOCATE** – Bellekten kaldır

Cursor Tipleri Nelerdir?

Cursor tipi, verinin nasıl gezileceğini belirler. İşte 4 temel tip:

Cursor Tipi	Açıklama
STATIC	Veriyi bir kopya alır. Orijinal tabloda değişiklik olursa etkilenmez.
DYNAMIC	Anlık olarak tabloyu yansıtır. Veri değişirse CURSOR da güncellenir.
KEYSET-DRIVEN	Sadece satır anahtarları (key'ler) sabittir. Değerler değişebilir.
FAST_FORWARD	Sadece ileri yönde okur. Hızlıdır. Scroll yapamazsın.

static → veri kümesinin değişmemesi gerektiğinde kullanılır. veriler sabittir ve her zaman aynı sonucu verir. performansı dinamik cursor'a göre daha iyidir. dezavantajı ise veri kümesindeki değişiklikleri takip etmez.

```
/*
```

```
Bir Ogrenciler tablosu var.
```

```
STATIC cursor ile tüm öğrencileri dolaşacağız ve adlarını yazdıracağız.
```

```
Arada tabloda değişiklik yapsak bile cursor etkilenmeyecek.
```

```
*/
```

```
DECLARE @ad VARCHAR(50);
```

```
-- STATIC cursor tanımı
```

```
DECLARE static_cursor CURSOR STATIC FOR
```

```
SELECT Ad FROM Ogrenciler;
```

```
OPEN static_cursor;
```

```
FETCH NEXT FROM static_cursor INTO @ad;
```

```
WHILE @@FETCH_STATUS = 0
```

```
BEGIN
```

```
    PRINT 'Öğrenci Adı: ' + @ad;
```

```
    FETCH NEXT FROM static_cursor INTO @ad;
```

```
END
```

```
CLOSE static_cursor;
```

```
DEALLOCATE static_cursor;
```

```
/*
```

```
STATIC olduğu için SELECT Ad FROM Ogrenciler sonucu belleğe alınır.
```

```
O sırada biri gelip Ogrenciler tablosuna yeni öğrenci eklese bile bu cursor onları görmez.
```

```
Çünkü cursor anlık kopya ile çalışır.*/
```

```
INSERT INTO Ogrenciler (Ad) VALUES ('YeniÖğrenci');
```

```
/*
```

```
Yukarıdaki gibi cursor açıkken tabloya veri
```

ekleyebilirsin ama cursor hala eski veriyle çalışır. Yeni geleni almaz.

*/

dinamik→ **Canlı veri** ile çalışır: Orijinal tablodaki **her değişikliği anında yansıtır**. Okuma sırasında biri veri ekler/siler/güncellerse, cursor bunu görür. Çok esnek ama daha **fazla kaynak** tüketir. **Scroll yapılabilir** (ileri-geri, başa dön gibi hareketler mümkün).

Aşağıda, Ogrenciler tablosundaki adları dinamik olarak dolaşıyoruz.
Bu sırada tabloya yeni kayıt eklersen, cursor onu anında görecektir!

```
DECLARE @ad VARCHAR(50);
-- DYNAMIC cursor tanımı
DECLARE dyn_cursor CURSOR DYNAMIC FOR
SELECT Ad FROM Ogrenciler;
OPEN dyn_cursor;
FETCH NEXT FROM dyn_cursor INTO @ad;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Öğrenci Adı: ' + @ad;
    -- Örneğin burada yeni öğrenci eklersen:
    -- INSERT INTO Ogrenciler (Ad) VALUES ('CanlıOğrenci');
    FETCH NEXT FROM dyn_cursor INTO @ad;
END
CLOSE dyn_cursor;
DEALLOCATE dyn_cursor;
/*
Cursor çalışırken INSERT işlemi yap DYNAMIC olduğu için
cursor o satırı da görecektir.
Aynı şey DELETE veya UPDATE için de geçerli.*/
INSERT INTO Ogrenciler (Ad) VALUES ('DinamikEklenen');
```

STATIC vs DYNAMIC KARŞILAŞTIRMA TABLOSU

Özellik	STATIC CURSOR	DYNAMIC CURSOR	
Veri Güncelliği	Anlık kopya, değişiklikleri görmez	Canlı veri, değişiklikleri anında görür	
Performans	Daha hızlı ve düşük bellek kullanımı	Daha yavaş, daha fazla kaynak tüketir	
Kullanım Amaçları	Raporlama, analiz	Gerçek zamanlı kontrol, aktif sistemler	
İzleme Yeteneği	Sabit içerik	Satır ekleme/silme anında görünür	
Scroll Desteği	Evet	Evet	

forward_only→ **en hızlı** çalışan ama **sadece ileriye** doğru okuma yapabilen cursor'dur.
cursor'ın **sadece ileri yönlü okuma yapmasına** izin veren türüdür. Diğer cursor türlerinde ileri-

geri veya başa dönme gibi işlemler yapılabilirken, **FORWARD_ONLY**'de sadece bir kere ilerlenir.

Özellik	Açıklama
Yön	Sadece ileri (forward)
Performans	En hızlı cursor türlerinden biridir
Bellek Kullanımı	Düşük
Scroll Desteği	Yok (geri dönüş yok, satır atlanamaz)
Kullanım Amaçları	Basit, hızlı döngülerde tercih edilir

```
DECLARE @ad VARCHAR(50);

-- FORWARD_ONLY tanımlanması
DECLARE fwd_cursor CURSOR FORWARD_ONLY FOR
SELECT Ad FROM Ogrenciler;

OPEN fwd_cursor;
FETCH NEXT FROM fwd_cursor INTO @ad;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Öğrenci Adı: ' + @ad;
    FETCH NEXT FROM fwd_cursor INTO @ad;
END
CLOSE fwd_cursor;
DEALLOCATE fwd_cursor;
/*
Bu cursor tipi şunlar için idealdir:
Tüm öğrencilerin adı yazdırılacak ama geriye dönmeye gerek yok
İşlem hızlı ve hafif çalışmalı.
Yalnızca satır satır ileri gidilecek

FETCH PRIOR, FETCH LAST gibi işlemler hata verir.
Geri dönmek mümkün değildir, kaçan satırı tekrar yakalayamazsın.
*/
```

keyset → Bu cursor, hem **STATIC** hem **DYNAMIC** cursor'ların iyi yönlerini birleştirir gibi davranır. Hemen açıklayayım, sonra sınavlık örneğimizi yazalım. Cursor çalıştırıldığında **satırların anahtarları (keyset)** belleğe alınır. Bu sayede **satırların varlığı sabittir**, ama **veri değişiklikleri (güncelleme gibi)** yansıtılabilir.

Özellik	Açıklama
Anahtarlar	Belleğe alınır ve sabit kalır
Veri Değişimi	Değer değişiklikleri yansıtılır (UPDATE görünür), ama yeni satırlar görünmez

Satır Silinirse	#DELETED olarak görünür, ama sıradan çıkarılamaz
Yön	Scroll yapılabilir (ileri-geri)
Performans	STATIC ve DYNAMIC arasında ortalama

```

DECLARE @id INT, @ad VARCHAR(50);
DECLARE key_cursor CURSOR KEYSET FOR
SELECT OgrenciID, Ad FROM Ogrenciler;
OPEN key_cursor;
FETCH NEXT FROM key_cursor INTO @id, @ad;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'ID: ' + CAST(@id AS VARCHAR) + ' - Ad: ' + @ad;

    -- Güncelleme yapılabilir, satır silinirse #DELETED olur
    -- UPDATE Ogrenciler SET Ad = UPPER(@ad) WHERE OgrenciID = @id;

    FETCH NEXT FROM key_cursor INTO @id, @ad;
END
CLOSE key_cursor;
DEALLOCATE key_cursor;
/*
Bir satırı güncelle: Cursor bunu anında görecektir.
Yeni satır ekle: Cursor göremez.
Satır sil: Cursor satırı #DELETED olarak tanır ama çıkaramaz.

Satır kimlikleri sabit kalmalı ama veriler güncellenebilir olmalı
Hem esneklik hem güvenlik isteniyorsa
Satırların değişmesini engellemek, ama içeriğini takip etmek gerekiyorsa
*/

```

Cursor Türlerinin Özet Karşılaştırması:

Özellik	STATIC	DYNAMIC	KEYSET-DRIVEN	FORWARD_ONLY
Satır Sayısı	Sabit	Canlı	Sabit (ID sabit)	Sabit
Veri Değişimi	Yansımaz	Yansır	Değişiklik yansır	Yansımaz
Satır Ekleme	Görünmez	Görünür	Görünmez	Görünmez
Silinmiş Satır	Yok sayılır	Silinir	#DELETED görünür	Yok sayılır
Scroll	Evet	Evet	Evet	Hayır
Performans	Hızlı	Yavaş	Orta	En hızlı

SCROLLABLE CURSOR (Kaydırılabilir Cursor)

Nedir?

- Cursor içinde **satırlar arasında ileri, geri, belli bir satıra gitme (ABSOLUTE/RELATIVE)** gibi işlemleri yapmamıza olanak tanır.
- Default olarak **STATIC** , **KEYSET** , **DYNAMIC** tipleri **SCROLL destekler**.
- **SCROLL** ifadesi **özellikle belirtilirse** özel davranış gösterir.

Ne işe yarar?

- Sonraki (NEXT), önceki (PRIOR), ilk (FIRST), son (LAST), belirli satıra (ABSOLUTE X) gitmek mümkün.
- Özellikle **raporlama ve veri gezintisi (paging)** yapılacaksa kullanılır.

```
DECLARE @ad VARCHAR(50);
DECLARE sc_cursor CURSOR SCROLL FOR
SELECT Ad FROM Ogrenciler;
OPEN sc_cursor;
FETCH LAST FROM sc_cursor INTO @ad;
PRINT 'Son öğrenci: ' + @ad;
FETCH FIRST FROM sc_cursor INTO @ad;
PRINT 'İlk öğrenci: ' + @ad;
FETCH PRIOR FROM sc_cursor INTO @ad;
-- Başta olduğumuz için hata verebilir
CLOSE sc_cursor;
DEALLOCATE sc_cursor;
```

FAST_FORWARD CURSOR

💡 **Nedir?**

- En **hızlı** çalışan cursor tipidir çünkü:
 - **Sadece ileri yönlü** (**FORWARD_ONLY**)
 - **Güncelleme desteklemez (READ-ONLY)**
- Belirli bir işlemi **çok hızlı geçmek** istiyorsan kullanılır.

Ne işe yarar?

- Tüm tabloyu satır satır hızlıca dolaşmak istiyorsan idealdir.
- **READ ONLY** olduğundan veriyi **güncelleyemezsin, silemezsin**.
- **SCROLL** , **UPDATE** , **DELETE** desteklemez.

```
DECLARE @ad VARCHAR(50);

DECLARE ff_cursor CURSOR FAST_FORWARD FOR
SELECT Ad FROM Ogrenciler;
OPEN ff_cursor;
```

```

FETCH NEXT FROM ff_cursor INTO @ad;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Ad: ' + @ad;
    FETCH NEXT FROM ff_cursor INTO @ad;
END
CLOSE ff_cursor;
DEALLOCATE ff_cursor;

```

KISA ÖZET (Karşılaştırma):

Özellik	SCROLLABLE	FAST_FORWARD
Satır Yönu	İleri, geri, rastgele	Sadece ileri
Performans	Orta	En hızlı
Güncelleme	Destekler (cursor tipine bağlı)	DESTEKLEMEZ (READ ONLY)
Kullanım Alanı	Raporlama, gezinti	Seri, hızlı işlemler
Bellek Kullanımı	Fazla olabilir	Az

Ne Zaman Hangi Cursor?

- **STATIC** → Raporlamada, değişmeyen veriyle çalışırken
- **DYNAMIC** → Canlı sistemlerde veri değişimi takip edilmek isteniyorsa
- **KEYSET** → Satırlar belli ama değerler değişebilir durumlar
- **FAST_FORWARD** → Hızlı tek yönlü geçiş yeterliyse (en çok kullanılan!)

DİKKAT! Cursor performans açısından **zayıftır**. Çok büyük verilerde **döngü yerine set-based (tek seferde işlem)** mantığı tercih edilir.

örnek

Bu örnekte, bir öğrenciler tablosundan

her öğrencinin adını ekrana yazdıracağız ve adında "a" harfi geçen öğrencilerin ismini büyük harfe çevirerek ayrı bir tabloya ekleyeceğiz

- **Ogrenciler** tablosunda **OgrenciID**, **Ad**, **Soyad**, **Durum** sütunları var.
- İçinde "a" harfi olan adlar **OgrenciLog** adlı ayrı tabloya loglanacak.

```

-- İLK ÖNCE YARDIMCI TABLOYU OLUŞTURALIM.
CREATE TABLE OgrenciLog (
    OgrenciID INT,
    BuyukAd VARCHAR(50)
);
-- CURSORLI İŞLEMİ YAPTIRALIM
DECLARE @id INT, @ad VARCHAR(50);
DECLARE ogrCursor CURSOR FOR
SELECT OgrenciID, Ad FROM Ogrenciler;
OPEN ogrCursor;
FETCH NEXT FROM ogrCursor INTO @id, @ad;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Öğrenci Adı: ' + @ad;
    IF @ad LIKE '%a%'
    BEGIN

```

```

INSERT INTO OgrenciLog (OgrenciID, BuyukAd)
VALUES (@id, UPPER(@ad));
END
FETCH NEXT FROM ogrCursor INTO @id, @ad;
END
CLOSE ogrCursor;
DEALLOCATE ogrCursor;

```

Kod Parçası	Ne işe yarıyor?
DECLARE ogrCursor CURSOR FOR...	Cursor tanımlanıyor
FETCH NEXT FROM ... INTO ...	Her satırı sırayla alıyor
IF @ad LIKE '%a%'	Adında "a" olanları tespit ediyor
UPPER(@ad)	Adı büyük harfe çeviriyor
INSERT INTO OgrenciLog...	Log tablosuna ekliyor

- Cursor'lar set-based işlemler yerine kullanılabilir, ancak büyük veri kümeleri üzerinde cursor kullanımı genellikle performans kayıplarına neden olur. Bunun yerine, mümkünse set-based SQL sorguları kullanmak daha verimli olacaktır.
- Ayrıca, cursor kullanıldığında dikkat edilmesi gereken önemli noktalar:

Veritabanı üzerindeki locking (kilitleme) etkilerini göz önünde bulundurmak gerekir.

Resource tüketimi (bellek, işlemci gücü) artabilir.

Cursor kullanımı gerektiren durumlarda en uygun cursor tipi seçilerek işlem yapılmalıdır.

YEDEKLEME YÖNETİMİ

yedekleme türleri;

tam yedekleme; veritabanının tam yedeğini alır.

fark yedeği, en son tam yedekten sonra değişen verilerin yedeğidir.

işlem günlüğü yedeği; tüm işleme geçmişinin yedeğidir. veri tabanının belirli zamana geri yüklenmesini sağlar.

dosya ve dosya grubu yedekleme; büyük veritabanlarında kullanılır belli dosyaların yedeklenmesini sağlar.

tam yedekleme;;

sqlserver için en temel yedekleme türüdür. tüm veritabanlarını içerir. restore sırasında farklı ve işlem günlüğü yedekleriyle birlikte kullanılabilir.

```

BACKUP DATABASE Okul
TO DISK = 'C:\yedek\OkulFull.bak'
WITH INIT;

```

differential yedekleme;

en son alınan tam yedekten sonra değişen tüm verilerin yedeğini alır. daha hızlıdır ve daha az disk alanı kullanır. ancak tek başına geri yükleme yapılmaz.

```
BACKUP LOG Okul  
TO DISK = 'C:\yedekek\OkulLog.trn';
```

◆ Yedekleme Zamanlaması

- Tam Yedekleme: Her gece saat 02:00
- Differential Yedekleme: Her gün 12:00
- Log Yedekleme: Her 15 dakikada bir
- Otomasyon: SQL Server Agent ile görev zamanlaması yapılabilir.

◆ Yedekleme Dosyalarının Yönetimi

- Dosya isimlerinde tarih/zaman bilgisi kullanılmalı
 - Örnek: Okul_20240601.bak
- Disk doluluk kontrol edilmeli
- Eski yedekler düzenli olarak temizlenmeli
- Gerekirse otomatik silme (Maintenance Plan) yapılmalı

◆ Maintenance Plan ile Yedekleme

- SQL Server Management Studio'da "Maintenance Plan" aracı ile:
 - Otomatik yedekleme
 - Otomatik temizlik
 - Otomatik e-posta bildirimi yapılabilir.

◆ Örnek Maintenance Plan

- Adım 1: Full Backup
- Adım 2: Differential Backup
- Adım 3: Transaction Log Backup
- Adım 4: Cleanup Task (7 günden eski yedekleri sil)

◆ Yedekleme Stratejisi Örneği

- Pazartesi - Pazar
 - Tam yedek: Pazar gece
 - Fark yedeği: Pazartesi - Cumartesi arası her gece
 - Log yedeği: Her saat başı

◆ Yedekleme Hataları ve Önlemler

- Yetersiz disk alanı
- Yanlış yedekleme yolu
- Erişim izinleri eksikliği
- Çözüm:
 - Disk alanı takibi
 - Doğru klasör yetkileri
 - SQL Agent izni kontrolü

◆ Yedekleme Geri Yükleme Senaryosu

- Ani veri kaybı yaşanırsa:
 1. Tam yedek geri yüklenir (NORECOVERY)
 2. Fark yedeği geri yüklenir (NORECOVERY)
 3. Tüm işlem günlükleri sırasıyla yüklenir
 4. Son log yedeği **WITH RECOVERY** ile tamamlanır

Sonuç

- Yedekleme veritabanı yönetiminin en kritik adımıdır.
- Uygun strateji ile veri kayıpları önlenir.
- Düzenli testler ile yedeklerin sağlamlığı kontrol edilmelidir.
- Otomasyon ve düzenli planlama ile süreç güvence altına alınır.

1-Stok Yönetim Sistemi

Durum: Şirketinin bir **StokYonetim** adlı MSSQL veritabanı var. Bu veritabanında: **Ürünler (Urunler), Siparişler (Siparisler), Depolar (Depolar)** gibi tablolar mevcut.

Bu veritabanı şirket için kritik; **günlük işleyişi etkileyecek bir sorun yaşanmaması için düzenli yedekleme şart.**

Yedekleme Stratejisi:

Her gece 02:00'de tam yedek alınır.

Her saat başı differential yedek alınır.

Transaction log yedeği 15 dakikada bir alınır.

```
--Tam Yedek Alma (Gece 02:00 Yedeği)
BACKUP DATABASE StokYonetim
TO DISK = 'C:\Backup\StokYonetim_FULL.bak'
WITH FORMAT, MEDIANAME = 'StokYonetimFull', NAME = 'StokYonetim Tam Yedek';

--Differential Yedek Alma (Saat Başlarında)
BACKUP DATABASE StokYonetim
TO DISK = 'C:\Backup\StokYonetim_DIFF_1000.bak'
WITH DIFFERENTIAL, NAME = 'StokYonetim 10:00 Differential Yedek';

--Transaction Log Yedeği (15 Dakikada Bir)
BACKUP LOG StokYonetim
TO DISK = 'C:\Backup\StokYonetim_LOG_1015.trn'
WITH NAME = 'StokYonetim 10:15 Log Yedeği';
```

Felaket Kurtarma Senaryosu (Restore)

a) Tam Yedeği Geri Yükle:

```
RESTORE DATABASE StokYonetim
FROM DISK = 'C:\Backup\StokYonetim_FULL.bak'
WITH NORECOVERY;
```

b) Differential Yedek:

```
RESTORE DATABASE StokYonetim
FROM DISK = 'C:\Backup\StokYonetim_DIFF_1000.bak'
WITH NORECOVERY;
```

c) Log Yedeği:

```
RESTORE LOG StokYonetim
FROM DISK = 'C:\Backup\StokYonetim_LOG_1015.trn'
```

```
WITH RECOVERY;
```

2–Senaryo: Finans Departmanının Veri Güvenliği Stratejisi

Durum: Bir şirkette **Finans departmanı**, kritik öneme sahip **FinansDB** adlı bir veritabanı kullanıyor. Bu veritabanında maaş ödemeleri, giderler, banka hareketleri gibi hassas veriler tutuluyor.

Amaç:

Her gece tam yedek alınacak. Gün içinde saat başı işlem günlükleri (log backup) alınacak. Haftalık olarak fark yedekleri alınacak. Yedekler, ayrılmış bir klasöre yazılacak ve **7 gün sonra otomatik silinecek**.

Çözüm ve Yedekleme Planı

1. Veritabanı Yedekleme Modunun Ayarlanması

```
ALTER DATABASE FinansDB SET RECOVERY FULL;
```

2. Tam Yedekleme (Her gece 02:00)

```
BACKUP DATABASE FinansDB  
TO DISK = 'D:\SQLBackups\FinansDB_Full.bak'  
WITH FORMAT, INIT, NAME = 'Full Backup - FinansDB';
```

3. Fark Yedeği (Her Cumartesi 12:00)

```
BACKUP DATABASE FinansDB  
TO DISK = 'D:\SQLBackups\FinansDB_Diff.bak'  
WITH DIFFERENTIAL, NAME = 'Differential Backup - FinansDB';
```

4. Log Yedeği (Saatlik olarak)

```
BACKUP LOG FinansDB  
TO DISK = 'D:\SQLBackups\Logs\FinansDB_Log.trn'  
WITH INIT, NAME = 'Log Backup - FinansDB';
```

5. Eski Yedekleri Otomatik Silme (Maintenance Plan ile)

"Maintenance Cleanup Task" eklenir.

7 günden eski **.bak** ve **.trn** dosyaları ***D:\SQLBackups*** klasöründen silinir.

```
--OLASI GERİ YÜKLEME SENARYOSU (restore)  
RESTORE DATABASE FinansDB  
FROM DISK = 'D:\SQLBackups\FinansDB_Full.bak'
```

```
WITH NORECOVERY;
```

```
RESTORE DATABASE FinansDB  
FROM DISK = 'D:\SQLBackups\FinansDB_Diff.bak'  
WITH NORECOVERY;
```

```
RESTORE LOG FinansDB  
FROM DISK = 'D:\SQLBackups\Logs\FinansDB_Log.trn'  
WITH RECOVERY;
```

BÜYÜK VERİ KULLANIMI

Büyük veri, geleneksel veritabanı sistemlerinin işleyemeyeceği kadar **hızlı, hacimli ve çeşitli verilerin** bir araya geldiği veri kümeleridir. Bu veriler yapılandırılmış (structured), yapılandırılmamış (unstructured) veya yarı yapılandırılmış (semi-structured) olabilir.

Özellik	Açıklama
Volume (Hacim)	Büyük miktarda veridir. GB, TB, hatta PB seviyesinde olabilir.
Velocity (Hız)	Veri çok hızlı üretilir ve işlenmelidir. Gerçek zamanlı işleme gerekir.
Variety (Çeşitlilik)	Metin, ses, video, log, JSON, XML gibi çok farklı formatlarda gelir.
Veracity (Doğruluk)	Verinin doğruluğu, güvenilirliği önemlidir. Gürültülü veri barındırabilir.
Value (Değer)	Veriden iş zekâsı, öngörü veya karar destek için anlamlı bilgi çıkarılır.

!!!burada slaytlarda olmayan doğruluk ve değer adında iki özellikle daha ekledim.

GERÇEK HAYATTAN BÜYÜK VERİ ÖRNEKLERİ

◆ 1. İlaç Sektörü – Türkiye 2017 Satış Verisi

- Her barkod okutma bir veri noktasıdır.
- Türkiye genelinde günde milyonlarca barkod okutulur.
- Bu veriler reçete geçmişi, satış analizleri, depolama gereksinimleri gibi alanlarda kullanılır.

◆ 2. Sosyal Medya

- Twitter: Her saniye binlerce tweet.
- Instagram: Her dakika binlerce fotoğraf, etkileşim.
- Bu verilerin işlenmesi için dağıtık sistemler gerekir (Apache Hadoop, Spark vb.)

◆ 3. E-Ticaret Siteleri

- Kullanıcı tıklamaları, sepete eklemeler, satın alma davranışları.
- Veriler hem öneri sistemlerinde hem stok analizlerinde kullanılır.

SQL Server ile Büyük Veri Ne Alaka?

- SQL Server 2019 ile birlikte **Big Data Cluster** desteği geldi.
- **PolyBase** ile Hadoop dosyalarını SQL Server üzerinden sorgulamak mümkün.
- **Index, partition, in-memory** gibi yöntemlerle büyük veri optimize edilir.

ETL nedir?

ETL, Extract (Çek), Transform (Dönüştür), Load (Yükle) kelimelerinin baş harflerinden oluşur. Verilerin bir kaynaktan alınıp, işlenerek farklı bir hedef sisteme aktarılmasını sağlayan veri entegrasyon sürecidir. Özellikle

büyük veri sistemlerinde dağınık ve çok kaynaklı verilerin temizlenerek anlamlı hale getirilmesi için ETL hayati önem taşır.

ETL, veriyi bir kaynaktan alıp (Extract), dönüştürüp (Transform), hedef sisteme (Load) yükleme sürecidir. Bu süreç veri ambarlarını beslemek için kullanılır.

Aşama	Açıklama
Extract	Farklı kaynaklardan veri çekilir (veritabanı, API, Excel).
Transform	Temizleme, tür dönüştürme, iş kurallarının uygulanması.
Load	Veri ambarına yazılır (genelde sadece eklenir).

1.Extract (Veri Çekme)

- Veri, ilişkisel veritabanı (SQL Server, Oracle), CSV, Excel, API, IoT sensörleri, sosyal medya gibi **bir veya birden fazla kaynaktan** alınır.
- Amaç: Kaynak sistemdeki veriyi **bozmadan** alma.

```
SELECT * FROM Satinalma.dbo.Musteriler;
```

2. Transform (Veri Dönüştürme)

Alınan veri temizlenir, filtrelenir, dönüştürülür. Null değerler yönetilir, tip dönüşümleri yapılır. Kodlama işlemleri (örneğin: ülke kodunu ülke adına çevirme), gruplama, özetleme işlemleri yapılır.

```
SELECT
  ISNULL(Ad, 'Bilinmeyen') AS MusteriAdi,
  CAST(Tarih AS DATE) AS SiparisTarihi,
  CASE
    WHEN Tutar > 1000 THEN 'VIP'
    ELSE 'Standart'
  END AS MusteriTipi
FROM Siparisler;
```

3. Load (Veri Yükleme)

Dönüştürülmüş veri hedef sisteme (Data Warehouse, analiz motoru, BI aracı) yüklenir.

Bu işlem **tam (overwrite)** veya **ekleme (append)** şeklinde olabilir.

```
INSERT INTO DataWarehouse.dbo.TemizSiparisler
SELECT * FROM StagingArea.dbo.SiparisTemiz;
```

özelliklerine gelecek olursak da

Özellik	Açıklama
Tarihsel veri işleme	Zaman boyutlu veri beslemesi
Otomasyon	SQL Agent veya SSIS ile zamanlanabilir
Doğrulama ve hata yönetimi	TRY...CATCH veya SSIS Error Output kullanılır

🧠 ETL Nerelerde Kullanılır?

- **Eczane Barkod Simülasyonları**
→ Her barkod verisi önce geçici tabloda temizlenip sonra ana tabloya aktarılır.
- **Müşteri Bilgi Entegrasyonu**
→ CRM'den gelen veriler ETL ile filtrelenip satış veritabanına yüklenir.
- **Raporlama Sistemleri**
→ Gün sonunda toplanan işlem verileri, sabah rapor sistemine yüklenir.

🔗 ETL ile Farkı Ne?

Özellik	ETL	ELT
Dönüşüm nerede yapılır?	Ara katmanda (ETL aracı)	Hedef sistemde (örneğin: SQL Server)
Hız	Yavaş ama kontrol edilebilir	Daha hızlı, paralel
Büyük veri desteği	Kısıtlı	Uygun (örneğin: Azure, Hadoop)
Kullanıldığı yerler	Geleneksel sistemler	Cloud, Data Lake sistemleri

SQL SERVER'DA ETL MİMARİSİ

kullanılabilecek yöntemler

Yöntem	Açıklama
T-SQL ETL	Manuel SQL komutlarıyla veri çekme, dönüştürme, yazma
SSIS	SQL Server Integration Services – GUI tabanlı ETL paketi
OPENROWSET	Dosya, uzak veritabanı gibi kaynaklardan direkt veri çekme
Linked Server	Başka bir SQL Server'a bağlı sorgularla veri aktarımı

🔗 SSIS ile ETL Süreci Örneği:

Adımlar:

1. Data Flow Task → Kaynak veritabanı seçilir.
2. Derived Column → Yeni sütun hesaplanır.
3. Conditional Split → Veriler gruplandırılır.
4. Destination → Veri ambarına yazılır.

📌 SSIS, sürük-le-bırak ile görsel ortamda işlem yapmaya olanak tanır.

SUNUM KATMANI NEDİR?

Sunum katmanı, kullanıcıların raporlama araçları ile etkileşime geçtiği kısımdır.

SQL Server tarafında **View** , **Indexed View** ,**Tabular Model**, **SSRS** veya dışarıdan **Power BI** kullanılır.

SQL View ile Sunum Katmanı

```
CREATE VIEW vw_UrunSatisRaporu AS
SELECT
    U.UrunAdi,
    M.Sehir,
    Z.Yil, Z.Ay,
    SUM(S.Tutar) AS ToplamSatis
FROM Satislar S
JOIN Urun U ON S.UrunID = U.UrunID
JOIN Musteri M ON S.MusteriID = M.MusteriID
JOIN Zaman Z ON S.ZamanID = Z.ZamanID
GROUP BY U.UrunAdi, M.Sehir, Z.Yil, Z.Ay;
```

--Bu View, Power BI'da bağlanarak dinamik rapor haline getirilebilir.

Slayt	Eksik	Bu içerikte neyle tamamlandı
Yağmur PDF / Zeynep-Şevval PDF	Star schema/snowflake tanımı	Yapı + görselleştirme ilişkisi eklendi
Veri Sunumu	Sunum katmanı tanımı	SQL View ile örnek View + Power BI senaryosu
ETL Sunumu	Veri ambarı kavramı	Mimarideki yeri tanımlandı

eczaneler de milyonlarca barkod okutma sistemi gerçekleştirilir ve bu barkod verileri hangi ilaç ne zaman kim tarafından alındı gibi detaylarla devasa veri kümeleri oluşturur. İşt ebu yapı big data kısmına giriyor.


Kullanım Senaryoları:

1. Reçete Takibi ve Hatalı İlaç Tespiti

- Aynı hastaya farklı eczanelerden benzer reçete yazılması gibi durumlar tespit edilebilir.

2. Stok Yönetimi ve Talep Tahmini

- Hangi ilaç ne zaman çok satılıyor?
- Mevsimsel olarak hangi ilaçlara talep artıyor?

 Bu verilerle ilaçların sipariş takibi ve stok yenileme zamanı otomatik hesaplanabilir.

3. İlaç İsrafının Önlenmesi

- Son kullanma tarihi yaklaşan ilaçlar tespit edilir.
- Envanterdeki atıl kalan ilaçlar başka şubeye yönlendirilebilir.

4. Kampanya ve Öneri Sistemleri

- Hasta geçmişine göre "şunu da alabilirsiniz" gibi sistemler öneri sunabilir.
- Market tarzı eczanelerde bu büyük veriyle satış yönlendirmesi yapılabilir.

Toplanan Veri Türleri:

Veri Alanı	Açıklama
Barkod No	Ürünün kimliği
Satış Tarihi	Hangi tarihte işlem yapılmış
Hasta ID / TCKN	Anonimleştirilmiş hasta bilgisi
Eczane Kodu	Hangi eczaneden satılmış
Satış Fiyatı	İlaç indirimi, SGK katkısı
Reçete Türü	Normal / e-reçete / kırmızı reçete

Büyük Veri Ekosistemine gelecek olursak da; verinin toplandığı, işlendiği, analiz edildiği ve sunulduğu tüm sistemin bütünüdür.

temel bileşenleri;

Katman	Teknoloji Örnekleri	Görevi
Veri Kaynağı	Web, IoT, barkod sistemleri, sensörler	Veriyi üretir
Toplama	Apache Kafka, Flume	Gerçek zamanlı veri akışı
Depolama	Hadoop HDFS, Azure Data Lake, S3	Yüksek hacimli veri saklama
İşleme	Apache Spark, Hive, MapReduce	Büyük veri üzerinde dönüşüm ve analiz
Veri Ambarı	Snowflake, Azure Synapse, BigQuery	Analiz için organize veri
Sunum	Power BI, Tableau, SSRS	Görselleştirme ve raporlama

Neden böyle bir ekosistem gerekli?

çünkü: tek bir sistem bu kadar büyük veriyi saklayamaz ve tek başına işleyemez. bu yüzden her aşamayı yöneten ayrı sistemler birlikte çalışır. dağınık sistem yaklaşımı kullanılır.sql server da bu sisteme uyumlu hale geldi. polybase ile hadoop dosyasına bağlanabilir. linked server ile azure gibi sistemlerle konuşabilir.

DATA WAREHOUSE (Veri Ambarı) Nedir?

Veri ambarı, farklı kaynaklardan gelen verilerin **birleştirildiği, tarihsel olarak saklandığı** ve genellikle analiz için kullanıldığı özel bir veritabanıdır.

Özellikleri

- Genellikle **okuma odaklıdır** (OLAP - Online Analytical Processing).
- **Zaman boyutu** vardır (örnek: sipariş_tarihi, yıl, çeyrek).
- ETL sonrası **temiz, bütünleşmiş, anlamlı veri** içerir.
- Genellikle **Star Schema** veya **Snowflake Schema** ile modellenir.

☀ Veri Ambarı Mimarisi Katmanları

◆ 1. Kaynak Katmanı (Source Layer)

Veri CRM, ERP, Excel, log dosyaları gibi farklı sistemlerden gelir.

◆ 2. ETL Katmanı

Veriler dönüştürülüp veri ambarına aktarılır.

◆ 3. Veri Ambarı Katmanı

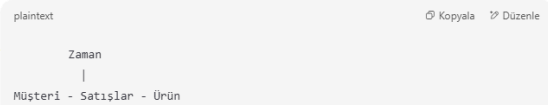
Veriler **tarihsel ve organize** şekilde burada saklanır (örneğin: Siparişler, Müşteriler).

◆ 4. Sunum Katmanı (Presentation Layer)

👉 Analiz, raporlama ve karar destek sistemlerine veri sağlar.
Excel, Power BI, SSRS gibi araçlarla görselleştirme yapılır.

🌟 Star Schema ve Sunum Katmanı İlişkisi

★ Star Schema Nedir?



- **Satışlar (Fact Table):** Sayısal veri içerir → adet, tutar, indirim

- **Müşteri, Zaman, Ürün (Dimension Tables):** Açıklayıcı nitelikler

Sunum katmanında bu şema sayesinde kullanıcılar:

- "2024 2. çeyrekte Ankara'da satılan ürün sayısı" gibi analizler yapar.
- Raporlamada slice/dice, drill-down işlemleri yapılabilir.

- **Snowflake Schema:** Dimension tablolar da kendi içinde parçalanır (örnek: İl → İlçe → Mahalle).
- **Star Schema** daha performanslıdır, raporlama için uygundur.

veri ambarı (data warehouse) geçmişe yönelik, konuya özel, bütünleştirilmiş ve karar destek amaçlı **analiz verilerinin** saklandığı özel bir veritabanı sistemidir.

Yani; operasyon verilerinden farklı olarak, verileri analiz ve raporlama için optimize eder.

Özellik	Açıklama
Konu Odaklı (Subject-Oriented)	Satış, hasta, ilaç, müşteri gibi konulara özel tablolar içerir.
Zamana Duyarlı (Time-Variant)	Geçmiş veriler değişmeden tutulur. Zaman boyutu vardır (Yıl, Ay).
Bütünleştirilmiş (Integrated)	Farklı sistemlerden gelen veriler ortak yapıda saklanır.
Kalıcı (Non-Volatile)	Veriler genellikle silinmez, sürekli genişler.

veritabanı ve veri ambarı arasındaki farklar ise;

Özellik	Veritabanı (OLTP)	Veri Ambarı (OLAP)
Amaç	Veri girişi, iş operasyonları	Karar destek, analiz
Veri yapısı	Normalleştirilmiş (3NF)	Denormalize (Star Schema)
Sorgu türü	Kısa ve sık sorgular (INSERT, UPDATE)	Uzun ve toplu sorgular (JOIN, SUM)
Veri güncelliği	Gerçek zamanlı	Tarihsel, sabit
Kullanıcı tipi	Operasyonel personel	Veri analisti, yönetici
Veri güncelleme	Sık güncellenir	Sadece eklenir, silinmez

Örnek Senaryo:

Veritabanı (OLTP):

Hasta eczaneye geldiğinde ilacını alır, veritabanına o an kaydedilir.

Veri Ambarı (OLAP):

"2023 yılı boyunca reçeteyle satılan antibiyotiklerin şehir bazlı satışı" raporu çıkarılır.

VERİ AMBARI MİMARİSİ

Temel Katmanlar:

1. Veri Kaynağı Katmanı:

- SQL, ERP, dosyalar, sensör verisi

2. ETL Katmanı:

- Temizleme, dönüştürme, eşleştirme işlemleri
- (SSIS, T-SQL, Python, Talend gibi)

3. Ambar Katmanı (DW):

- Fact tablolar (sayısal veri)
- Dimension tablolar (açıklayıcı veriler)

4. Sunum Katmanı:

- View, SSRS, Power BI, Tableau

VERİ AKTARIM KOMUTLARI;

INSERT INTO

tekil veya toplu veri ekleme:

```
INSERT INTO Hedef (Ad, Soyad)
SELECT Ad, Soyad FROM Kaynak;
```

MERGE

hem insert hem update hem delete tek komutta

```
MERGE Hedef AS h
USING Kaynak AS k
ON h.ID = k.ID
WHEN MATCHED THEN
    UPDATE SET h.Ad = k.Ad
WHEN NOT MATCHED THEN
    INSERT (ID, Ad) VALUES (k.ID, k.Ad);
```

BULK INSERT

dosyadan büyük veri alımı:

```
BULK INSERT Hedef
FROM 'C:\veri.csv'
WITH (FIELDTERMINATOR = ',', ROWTERMINATOR = '\n');
```

HATA YÖNETİMİ VE LOG TUTMA

```
BEGIN TRY
    -- işlem
END TRY
BEGIN CATCH
    INSERT INTO LogTablosu VALUES (ERROR_MESSAGE(), GETDATE());
END CATCH
-- T-SQL TRY CATCH İLE HATA YAKALANABİLİR.
--SSIS'te Error Output ile hatalı kayıtları başka tabloya yönlendirme yapılır.
--Log Provider → Dosyaya, SQL Server'a veya Event Viewer'a log yazabilir.
```

PERFORMANS İYİLEŞTİRMELERİ

SQL Server'da büyük veri ile çalışırken veriler çok büyüdüğünde **sorgular yavaşlar**, işlem süreleri uzar. Bunu çözmek için bazı önemli teknikler uygulanır:

iyileştirme yöntemleri

Yöntem	Açıklama
Index Kullanımı	Veriye hızlı ulaşmak için kullanılır. CREATE INDEX ile yapılır.
Partitioning	Büyük tabloları bölerek işleme (örnek: yıllara göre bölme).
Columnstore Index	Büyük veri setlerinde sütun bazlı saklama (OLAP için mükemmel).
Temp Table / CTE	Alt sorgular yerine geçici yapı kullanarak belleği verimli kullanma.
SSIS Buffer Tuning	SSIS'te veri taşırken buffer boyutları optimize edilir.

--ÖRNEK COLUMSTORE INDEX

CREATE CLUSTERED COLUMNSTORE INDEX idx_Satislar

ON Satislar;

-- ÖZELLİKLE 100 MİLYON+ SATIRLIK ANALİZ SORGULARINDA BÜYÜK PERFORMANS ARTIŞI

--Table Partitioning

-- Tarihe göre veriyi yıllara böl

CREATE PARTITION FUNCTION pf_yillar (DATE)

AS RANGE LEFT FOR VALUES ('2022-12-31', '2023-12-31', '2024-12-31');

BİG DATA İLE SQL SERVER ENTEGRASYONU

SQL Server, büyük veri teknolojilerine entegre olabilecek şekilde geliştirildi.

POLYBASE NEDİR?

- SQL Server ile Hadoop, Azure Data Lake gibi dış kaynaklara **SQL komutlarıyla erişim sağlar**.
- CSV, Parquet gibi dosyaları sorgulayabiliriz.

```
SELECT * FROM EXTERNAL_TABLE_ILACVERILERI
```

LINKED SERVER

Uzak SQL Server ya da başka sistemlerle bağlantı kurar.

```
SELECT * FROM [ECZANE_SUNUCU].[VeriDB].[dbo].[Satislar];
```

OPENROWSET

Dosyadan veya başka kaynaktan veri okur (güvenlik izinleri gerekir):

```
SELECT * FROM OPENROWSET(  
    BULK 'C:\veriler\ilac.csv',  
    FORMAT='CSV', FIRSTROW=2  
) AS IlacVerisi;
```

SSIS + BIG DATA

SSIS paketleriyle büyük veriye (HDFS, Blob, Azure Data Lake) doğrudan bağlanabilirsin.

Hadoop bağlantısı için özel connector'lar kullanılır.

Konu	Ne Sağlar?
Index / Partition	Sorgu hızını artırır
Columnstore Index	OLAP rapor performansını artırır
PolyBase	Hadoop, Azure verisine SQL ile erişim
SSIS + Big Data	Harici veri kaynaklarına entegrasyon
OpenRowSet / Linked	Uzak sistemlerden veri çekme

SQL Server'da

Big Data odaklı bir sistem nasıl kurulur

Amaç: Farklı kaynaklardan gelen devasa verileri (eczane satışları, reçete takibi, ürün stokları...) toplayıp ETL ile temizleyip Veri ambarına yükleyip Üzerinden analizler, tahminleme, karar destek sistemleri oluşturmaktır.

MİMARİ BİLEŞENLERİ:

1. Veri Kaynakları (Input Layer)

SQL veritabanları

Excel, CSV dosyaları

Web servisleri (API)

IoT cihazları (örneğin: barkod okuyucular)

2. ETL Katmanı

Araç/Yöntem	Görevi
SSIS	Sürükle-bırak ile ETL işlemleri
T-SQL	Manuel veri temizleme, dönüştürme
OPENROWSET	Dosyalardan veri çekme
Linked Server	Uzak SQL sunucudan veri alma

◆ 3. Staging Area (Geçici Alan)

- İlk alınan ham veriler burada tutulur.
- Veri kalitesi kontrol edilir (NULL, hatalı veri ayıklanır).

```
sql

INSERT INTO Staging_Satis
SELECT * FROM OPENROWSET(...);
```

◆ 5. Sunum Katmanı (Presentation Layer)

Teknoloji	Amaç
Power BI	Dinamik raporlar
SSRS	Faturalar, özet raporlar
Excel	Yönetimsel tablo çıktı
SQL View	Analiz için hazır sorgular

```
sql

CREATE VIEW vw_EczaneRapor AS
SELECT
    Eczane.Ad,
    Z.Yil, Z.Ay,
    SUM(FS.Tutar) AS ToplamSatis
FROM FactSatis FS
JOIN DimEczane Eczane ON FS.EczaneID = Eczane.ID
JOIN DimZaman Z ON FS.ZamanID = Z.ID
GROUP BY Eczane.Ad, Z.Yil, Z.Ay;
```

◆ 4. Veri Ambarı Katmanı (DW Layer)

- **Fact tablolar:** Satış, Stok, İade
- **Dimension tablolar:** Zaman, Ürün, Lokasyon, Eczane

```
sql

-- Örnek fact tablosu
CREATE TABLE FactSatis (
    SatisID INT PRIMARY KEY,
    UrunID INT,
    EczaneID INT,
    ZamanID INT,
    Adet INT,
    Tutar DECIMAL(10,2)
);
```

🔧 6. HATA YÖNETİMİ + LOG

- SSIS paketi → error output ile hatalı kayıtları "LogSatisHatalari" tablosuna atabilir.
- T-SQL TRY...CATCH → hata mesajlarını loglar:

```
sql

BEGIN TRY
    -- ETL kodları
END TRY
BEGIN CATCH
    INSERT INTO LogTablo VALUES (ERROR_MESSAGE(), GETDATE());
END CATCH;
```

🔒 7. GÜVENLİK + PERFORMANS + BACKUP

Alan	Açıklama
Yetkilendirme	Kullanıcılara sadece sunum katmanı açılır
Indexleme	Fact tablolarına index kurulur
Partitioning	Veri yıllara, bölgelere göre bölünür
Backup Planı	ETL sonrası tam yedek alınır

🧠 Gerçekçi Senaryo Örneği

🔥 "Eczane Barkod Sistemi"

- Günde 1 milyon işlem → staging tablosuna akar
- SSIS ile her gece 03:00'te ETL yapılır
- Veri ambarı: `FactSatis`, `DimZaman`, `DimUrun`, `DimHasta`
- Power BI: "Haftalık satış analizi", "Antibiyotik tüketim trendi" raporları

Bu yapı ile tam anlamıyla **Big Data + SQL Server + ETL + BI** birleştirilmiş olur ✨
Slaytlarda bu yapı parça parça geçiyordu, ama burada **bütünlük olarak mimari kurduk**.

SQL PROFILE

SQL Server Profiler, SQL Server üzerinde çalışan olayları **canlı olarak izlemeye**, kaydetmeye ve analiz etmeye yarayan bir **performans izleme aracıdır**.

Amaç: Hangi sorgular yavaş çalışıyor? Kim, ne zaman, hangi işlemi yaptı Veritabanı neden kilitlendi? Trigger tetiklenmiş mi?

Profilер Ne Yapar?

Özellik	Açıklama
Gerçek zamanlı izleme	Sorgular çalıştıkça anında görüntüleme
Trace dosyası kaydı	Olayları .trc dosyasına kaydeder, sonra analiz edilebilir
Filtreleme	Sadece belirli kullanıcı, sorgu tipi, veritabanı izlenebilir
Performans takibi	En yavaş çalışan sorgular kolayca tespit edilir

Profilер'da İzlenebilecek Bazı Event'ler:

EventClass	Açıklama
SQL:BatchStarting	TSQL komutu çalışmaya başlarken
SQL:BatchCompleted	TSQL komutu tamamlandığında
RPC:Completed	Stored procedure çağrıları
Showplan XML	Execution plan görüntüleme
Deadlock Graph	Kilitlenme (deadlock) tespiti

🗨 Kullanım Adımları (GUI ile):

1. SQL Server Management Studio (SSMS) → Tools → SQL Server Profiler
2. Yeni Trace oluştur (New Trace)
3. Giriş yap → Trace Name belirle
4. Template seç:
 - **TSQL_Replay**, **TSQL_Duration**, **Standard**
5. Filtre uygula:
 - DatabaseName = 'EczaneDB'
 - Duration > 1000 ms
6. Çalıştır → İzlemeye başla
7. Gözlem yap, **.trc** dosyasına kaydet

🎨 Örnek Uygulama Senaryosu:

🔴 Senaryo:

Eczane uygulamasında son 2 gündür sistem yavaşlıyor.

🌱 İzleme planı:

- Profiler başlatılır.
- **Duration > 1000** olan sorgular filtrelenebilir.
- **ApplicationName**, **HostName**, **LoginName** gibi kolonlar takip edilir.
- En yavaş sorgular belirlenir.
- Bu sorgular üzerinde **index**, **query tuning** yapılır.

KOLONLAR NE ANLAMA GELİYOR;

Kolon	Açıklama
TextData	Sorgunun içeriği
Duration	Sorgunun çalıştığı süre (milisaniye)
CPU	Ne kadar CPU tükettiği
Reads	Kaç kez veri okundu (I/O yükü)
Writes	Veri yazma işlemi sayısı
LoginName	Sorguyu çalıştıran kullanıcı

Profilер ile Gelen Veriler Nerede Kullanılır?

- Performans analizi
- Güvenlik incelemesi (kim hangi sorguyu çalıştırdı)
- Uygulama geliştirme aşamasında yük testi
- SSIS paketlerinde performans darboğazlarını bulmak

FELAKET KURTARMA VE DETAYLI VERİTABANI YÖNETİMİ

Failover Cluster

- ◆ Aynı ağda birden fazla sunucu tek bir **sanal IP** üzerinden çalışır.
- ◆ Ana sunucu çökünce yedek olan devreye otomatik girer.
- ◆ Depolama paylaşımlı, genellikle aynı lokasyonda.

Özellikleri:

- Otomatik geçiş (failover)
- Heartbeat ile sunucular birbirini dinler
- Tek bir veri kopyası vardır

⚠ Slaytta: “sistem odası çökerse yetersiz” denmişti — doğru, çünkü hepsi aynı fiziksel ortamda.

Log Shipping

- ◆ Ana sunucudan **transaction log yedekleri** belirli aralıklarla alınıp, yedek sunucuya kopyalanır ve yüklenir.

Bileşen	Görev
Primary Server	Logları üretir
Secondary Server	Logları okur ve yedeği oluşturur
Monitor Server (ops.)	Durumu izler, raporlar

- ◆ Manuel failover gerekir (otomatik değildir).
- ◆ Yedek sunucu **read-only** çalışır.

Slayttaki örnek:

Uygulamalı log shipping ekranlarıyla anlatılmış.

Database Mirroring

- ◆ Principal – Mirror – (opsiyonel Witness)
- ◆ INSERT/UPDATE/DELETE işlemleri önce yedeğe sonra anaya yazılır (senkron).

Modlar:

- **High Safety:** Senkron, Witness ile otomatik geçiş yapılır

- **High Performance:** Asenkron, manuel geiş yapılır

Slayttaki zet:

Yedek **NORECOVERY** ile restore edilir, endpoint tanımlanır, mirroring başlatılır.

AlwaysOn Availability Groups

- ◆ Hem failover cluster'ı hem mirroring'i birleřtirir
- ◆ Primary ve birden ok Secondary node olur
- ◆ Otomatik failover mmkndr
- ◆ Yedek sunuculardan backup ve SELECT yapılabilir

Replikalar:

Primary Only

Secondary Only

Any Replica

Prefer Secondary

Simlasyon slaytı (Fatih Orhan):

Logo Tiger ile senkron test yapıldı → 10 sn'de yeniden eřitlenme gzlemlendi.

DETAYLI VERİTABANI YÖNETİMİ

Roller ve Yetkiler:

```
CREATE LOGIN yedek_user WITH PASSWORD = '123';
CREATE USER yedek_user FOR LOGIN yedek_user;
EXEC sp_addrolemember 'db_backupoperator', 'yedek_user';
```

Maintenance Plan:

Yedek alma (full, diff, log)

Index rebuild / reorganize

Veri temizlięi (shrink, delete)

SQL Server Agent ile **zamanlanabilir**

İzleme:

sp_who2 → Aktif sorgular

sys.dm_exec_requests → Yavaş alışan işlemler

SQL Profiler → Hangi kullanıcı ne alıştırdı

Yöntem	Failover	Otomatik Geiş	Lokasyon	Okuma Yedeęi	Lisans
--------	----------	----------------	----------	--------------	--------

Failover Cluster	Evet	Evet	Aynı merkez	Hayır	Yüksek
Log Shipping	Hayır	Hayır (manuel)	Uzak	Evet (readonly)	Orta
Mirroring	Evet	Opsiyonel	Aynı / yakın	Hayır	Orta
AlwaysOn	Evet	Evet	Farklı lokasyonlar	Evet	Yüksek