# SQL SERVER TRANSACTION

16

Veritabanı sunucu, birçok işlemi doğru, eş zamanlı ve performanslı sorgulamak için optimize edebilecek yetenekler ile geliştirilmektedir. Veritabanı, sistem üzerinde belli kaynaklara sahiptir. Ve bu kaynakların sunduğu imkanlar, hız ve işlem gücü kapasitesi ile sınırlıdır. Bu işlem gücü ile gerçekleştirilen sorgular çok kısa sürelerde gerçekleştirilse de, zaman açısından belli bir saniye ya da salise gibi sürelerde gerçekleşir.

Bir sorgu içerisinde kullanılan satırlara başka kullanıcılar ya da sorgular da ulaşmak ve üzerinde değişiklik yapmak isteyebilir. Küçük çaplı uygulamalarda bu tür durumlar sorun oluşturmayabilir. Ancak bankacılık ya da benzeri büyük veri işlemleri gerçekleştiren çok istemcili veritabanı mimarilerinde aynı anda çok yoğun sorgular gerçekleştiriliyor olabilir.

Bir bankanın ülke genelinde 300 şubesi ve bankamatik cihazı olduğunu düşünelim. Aynı anda banka hesabından havale işlemi gerçekleştiren ya da herkesin çok sık kullanacağı benzer işlemler gerçekleştiren kullanıcıların veritabanı sunucuları üzerinde oluşturduğu işlem yükünü tahmin edebilirsiniz.

Aynı anda gerçekleştirilen işlemler için veritabanı sunucusunda bir işlem sırası gerçekleştirilmesi gerektiğini düşünüyor olabilirsiniz. Ve bu konuda doğal olarak haklısınız. Bir ya da birkaç tablo üzerinde veri değiştirme işlemi gerçekleştirilecekse ve değişiklik yapmak isteyen sorgu sayısı dakikada milyonları buluyorsa, belirli kriterler oluşturulması ve işlemlerin belirli iş blokları halinde gerçekleşmesi gerekir.

Bu bölümde, bir ya da birden fazla yığını (*batch*) tek bir iş bloğu olarak hazırlamayı inceleyeceğiz. İş bloklarının her bir parçasının çalışmasını garanti edecek sorgular oluşturarak, herhangi bir hata olasılığında, işlemleri geri almayı ya da işlem başarıyla gerçekleştirildiyse, gerekli bilgilerin log ve gerçek veri tablolarına işlenmesini inceleyeceğiz.

# Transaction ve Ortak Zamanlılık

Bazı durumlarda, gerçekleştirilmek istenen sorgu birden fazla işlemin aynı anda yapılmasını gerektirebilir. Aynı anda gerçekleşmesi gereken işlemlerden bir ya da birkaçının gerçekleşmemesi halinde başarılı şekilde gerçekleşen işlemlerin de geri alınabilmesi gerekir. Bu durum atom olma (*atomicity*), diğer bir deyişle parçalanamaz olma özelliği anlamına gelir.

Bu durumu nesnel olarak anlatmak gerekirse, paketlenmiş bir kutu düşünelim. İçerisinde çok parçalı ve birleştirildiğinde anlamlı bir nesneye dönüşecek yap-boz bulunuyor olsun. Size verilen görev ise, bu yap-boz parçaların hepsini doğru bir şekilde birleştirmenizdir. Eğer birleştiremezseniz, parçalar üzerinde gerçekleştirdiğiniz birleştirme işlemlerini iptal ederek, tüm parçaları ilk haline geri döndürmenizdir. Bunun nedeni, tüm parçaların tek bir bütünü temsil etmesidir. Bu parçalardan herhangi biri yanlış kullanılırsa bütün kesinlikle doğru olmayacaktır. Aynı şekilde, parçaları tek başına da kullanamazsınız. Çünkü bu parçalar sırasıyla ve doğru birleştirildiğinde bir anlam ifade edecektir.

Örnek verdiğimiz paketlenmiş yap-boz iş bloğunun veritabanındaki karşılığı transaction'dır.

Bir transaction bloğu SQL Server tarafında şu şekilde ele alınır:

- Transaction bloğu başlatılır. Transaction içerisindeki tüm iş parçacıkları bir bütün olarak tamamlanmak ya da bütün olarak iptal edilmek zorundadır. 4 işlem parçası içeren bir transaction bloğu içerisinde 3 işlem gerçekleşse de 1 işlemde hata alınırsa başarılı olan diğer 3 işlem geri alınır ve blok geçersiz olur. Transaction bloğu otomatik ya da kullanıcı tarafından başlatılabilir. Transaction'ı başlatmak için **`BEGIN TRANSACTION`** ya da kısa olarak **`BEGIN TRAN`** kullanılabilir.
- Transaction bloğu içerisinde yapılan her bir işlem bittiğinde işlemin başarılı olup olmadığına bakılır. İşlem başarısız ise, geri alma (**`ROLLBACK`**) gerçekleştirilir. İşlem başarılı ise diğer işlemlere devam edilir.

- Transaction, içerisindeki işlemler başarılı olarak tamamlandığında `COMMIT` ile bitirilir. Başarısız olunursa `ROLLBACK` ile geri alma işlemi gerçekleştirilir. `ROLLBACK` kullanıldığında, tüm veriler transaction önceki haline geri döner.

Bir veritabanının yetenekleri veri üzerindeki hakimiyet ve mimarisi ile doğru orantılıdır. Veritabanı, veriler üzerinde değişiklik yaparken `ACID` kuralını sağlamalıdır.

**ACID** (***A**tomicity, **C**onsistency, **I**solation, **D**urability*) kurallarını inceleyelim.

## Bölünemezlik (Atomicity)

Transaction işleminin ana özelliği olarak açıkladığımız bölünemezlik prensibini yansıtır. Bir transaction bloğu yarım kalamaz. Yarım kalan transaction bloğu veri tutarsızlığına neden olur. Ya tüm işlemler gerçekleştirilir, ya da transaction başlangıcına geri döner. Yani transaction'ın gerçekleştirdiği tüm değişiklikler geri alınarak gerçekleşmeden önceki haline döner.

## Tutarlilik (Consistency)

Bölünemezlik kuralının alt yapısını oluşturduğu bir kuraldır. Transaction veri tutarlılığı sağlamalıdır. Yani bir transaction içerisinde güncelleme işlemi gerçekleştiyse ve ya kalan tüm işlemler de gerçekleşmeli ya da güncelle işlemi de geri alınmalıdır. Bu veri tutarlılığı açısından çok önemlidir.

## İzolasyon (Isolation)

Her transaction veritabanı için bir istek paketidir. Bir istek paketi (*transaction*) tarafından gerçekleştirilen değişiklikler tamamlanmadan bir başka transaction tarafından görülememelidir. Her transaction ayrı olarak işlenmelidir. Transaction'ın tüm işlemleri gerçekleştikten sonra bir başka transaction tarafından görülebilmelidir.

## Dayaniklilik (Durability)

Transaction'lar veri üzerinde karmaşık işlemler gerçekleştirebilir. Bu işlemlerin bütününü güvence altına almak için transaction hatalara karşı dayanıklı olmalıdır. SQL Server'da meydana gelebilecek sistem sorunu, elektrik kesilmesi, işletim sisteminden ya da farklı yazılımlardan kaynaklanabilecek hatalara karşı hazırlıklı ve dayanıklı olmalıdır.

Bilgisayar bilimlerinde imkansız diye bir şey yoktur. Bir sistemin hata vermesi yüzlerce sebebe bağlı olabilir. Örneğin; geçtiğimiz yıllarda büyük bir GSM operatörünün veri merkezinin bulunduğu binaya sel basması sonucu veri kaybı yaşamışlardı. Bu durum olasılığı düşük gibi görünebilir. Ancak bahsi edilen konu önemli veriler olunca, hesaplanması gereken olasılıkların sınırı yoktur. Tüm hata olasılıklarına karşı dayanıklı bir sistem geliştirilmelidir.

# Transaction Bloğu

SQL Server, diğer tüm büyük veritabanı yönetim sistemlerinde olduğu gibi veri işlemlerini birçok farklı katman ve işlem ile gerçekleştirir. Veritabanı motoru katmanında ve birçok alt servis tabanında gerçekleştirilen bu işlemler, sorguların ve verinin yönetilmesi için farklı iş süreçlerini yönetir.

SQL Server üzerinde herhangi bir veri değiştirme işleminde değişiklik anında veritabanına yansımaz. Üzerinde değişiklik yapılan sayfalar daha önce diskten hafızaya alınmamış ise öncelikle **tampon hafıza** (*buffer cache*) denilen hafıza bölgesine çağrılır. Hafıza bölgesindeki sayfalar üzerinde değişiklikler gerçekleştirilir ve veritabanına hemen yansıtılmaz. Hafızaya çağrılarak üzerinde değişiklik yapılan sayfalara **kirli sayfa** (*dirty page*) denir. Kirli sayfalarda tutulan değişiklikler gerekli iş süreçleri tamamlandıktan sonra veritabanına kaydedilir. Kirli sayfaların veritabanına kaydedilme işlemine ise **arındırma** (*flushing*) denir.

Veri tutarlılığı açısından ve veri kaybını önlemek için yapılan değişiklikler **.LDF** uzantılı transaction log dosyasına kaydedilir. Bu işlem, verinin veritabanına kaydedilmeden önce yapılması önemlidir. Bu log dosyaları disk üzerinde gerçek verinin bulunduğu veri sayfalarına yazılmadan önce, verinin kaybolmasını önlemek amacıyla kullanılır. Herhangi bir olası sorun karşısında, tutulan bu loglar, kaybolan verinin geri getirilmesini sağlar.

SQL Server log işlemleriyle ilgili yönetimi ve olası durumları bölümün ilerleyen kısımlarında detaylıca işleyeceğiz.

# Transaction İfadelerini Anlamak

Transaction yönetimi için kullanılan dört farklı ifade vardır. Bu ifadeler ile transaction başlatılabilir (`BEGIN`), işlemler geri alınabilir (`ROLLBACK`), transaction bitirilebilir (`COMMIT`) ya da kayıt noktaları (`SAVE`) oluşturulabilir.

## Transaction'ı Başlatmak: BEGIN TRAN

Transaction'ın başlangıcını belirtir. Bu kısımdan sonraki tüm işlemler transaction'ın bir parçasıdır. İşlem sırasında oluşabilecek olası sorunlarda geri alma ya da transaction'ın sonlandırılması gerçekleştirilebilir.

**Söz Dizimi:**

```
BEGIN TRAN[SACTION] [transaction_ismi | @transaction_degiskeni]
```

## Transaction'ı Tamamlamak: COMMIT TRAN

Transaction'ın tamamlandığını ve gerçekleştirilen transaction işlemlerinin kalıcı olarak veritabanına yansıtılması için kullanılır. Transaction tarafından etkilenen tüm değişiklikler, işlemlerin tamamı gerçekleşmese bile, bu işlemden sonra kalıcı hale gelir.

**COMMIT** işleminden sonra gerçekleşen değişikliklerin geri alınması için, bu işlemleri geri alacak yeni bir transaction oluşturulmalıdır. Örneğin; bir transaction ile, nümerik bir sütun üzerinde 10 birim azaltma işlemi yapıldı ise, bu işlemi geri almak için aynı sütun üzerinde 10 birim artırma işlemi yapacak yeni bir transaction oluşturulmalıdır.

**Söz Dizimi:**

```
COMMIT TRAN[SACTION] [transaction_ismi | @transaction_degiskeni]
```

## Transaction'ı Geri Almak : ROLLBACK TRAN

Transaction'ın gerçekleştirdiği tüm işlemleri geri almak için kullanılır. Yani, yapılan tüm işlemler transaction'ın başlangıcındaki haline geri döner. Verilerdeki değişikliklerin anında kalıcı olarak veritabanına yansıtılmadığını belirtmiştik. **ROLLBACK** ile gerçekleştirilen tüm işlemler geriye alınarak transaction sonucunun tutarlılığı garanti edilir.

**ROLLBACK** işlemi, oluşturduğunuz transaction mimarisine bağlı olarak, **kayıt noktalarına** (*save points*) geri dönüş için de kullanılabilir.

**Söz Dizimi:**

```
ROLLBACK TRAN[SACTION] [transaction_ismi | kayit_noktasi_ismi
| @transaction_degiskeni | @kayit_noktasi_degiskeni]
```

### Sabitleme Noktaları: SAVE TRAN

**ROLLBACK** işlemi transaction'da en başa dönmek için kullanılır. Bazen de belirli bir noktaya kadar gerçekleşen işlemlerin geçerli kalması istenebilir. Bu işlemlerden sonra gerçekleşecek işlemler için **ROLLBACK**'e ihtiyaç duyulabilir. Sabitleme noktaları oluşturulması, transaction içerisinde en başa dönmek yerine, belirlenen bir işlem noktasına dönmek için kullanılır.

**Söz Dizimi:**

```
SAVE TRAN[SACTION] [ kayit_noktasi_ismi | @kayit_noktasi_degiskeni ]
```

## Transaction Oluşturmak

Transaction işlemleri için en uygun örnekler bankacılık ve gsm operatörlerinin veritabanı işlemleridir. Karmaşık ve bir çok iş parçacığı ile gerçekleşen sayısız işlemden oluşan bu tür uygulamalarda iş bloklarının doğru planlanması ve kullanılması önemlidir.

Bir banka veritabanında kullanıcı hesaplarını tutan ve farklı iki banka hesabı arasında havale işlemini gerçekleştirmek için bir uygulama oluşturalım.

Banka müşterilerinin hesap bilgilerini tutan basit bir tablo oluşturalım.

```
CREATE TABLE Accounts(
 AccountID CHAR(10) PRIMARY KEY NOT NULL,
 FirstName VARCHAR(50),
 LastName VARCHAR(50),
 Branch INT,
 Balance MONEY
);
```

**Accounts** tablosuna kayıtlar girelim.

```
INSERT INTO Accounts
VALUES('0000065127','Cihan','Özhan', 489, 10000),
      ('0000064219','Kerim','Fırat', 489, 500);
```

Eklediğimiz kayıtları listeleyelim.

```
SELECT * FROM Accounts;
```

| | AccountID | FirstName | LastName | Branch | Balance |
|---|---|---|---|---|---|
| 1 | 0000064219 | Kerim | Fırat | 489 | 500,00 |
| 2 | 0000065127 | Cihan | Özhan | 489 | 10000,00 |

Artık müşteri hesapları arasında havale gerçekleştirebiliriz. Şimdi, havale için gerekli adımları inceleyelim.

Havale işlemi gerçekleştirecek bir prosedür için temel gereksinimler;

- Para gönderecek hesap sahibinin bilgisi.
- Parayı alacak hesap sahibinin bilgisi.
- Havale edilecek paranın miktarı.

İş süreçlerini, güvenlik ve yönetilebilirlik nedeniyle Stored Procedure'ler ile gerçekleştirmek doğru bir çözüm olacaktır.

```
CREATE PROC sp_MoneyTransfer(
 @PurchaserID CHAR(10),
 @SenderID  CHAR(10),
 @Amount  MONEY
)
AS
BEGIN
 BEGIN TRANSACTION
  UPDATE Accounts
  SET Balance = Balance - @Amount
  WHERE AccountID = @SenderID
 IF @@ERROR <> 0
 ROLLBACK
  UPDATE Accounts
  SET Balance = Balance + @Amount
```

```
  WHERE AccountID = @PurchaserID
 IF @@ERROR <> 0
 ROLLBACK
 COMMIT
END;
```

Cihan ÖZHAN isimli banka müşterisinin hesabında 10.000 birim, Kerim FIRAT isimli müşterinin ise 500 birim parası var.

Cihan ÖZHAN, Kerim FIRAT'a 500 birim para havale etmek istiyor. Oluşturulan prosedür ile havale işlemini gerçekleştirelim.

```
EXEC sp_MoneyTransfer '0000064219','0000065127',500;
```

Havale işleminden önceki ve sonraki hesap görünümü şu şekildedir.

| | AccountID | FirstName | LastName | Branch | Balance |
|---|---|---|---|---|---|
| 1 | 0000064219 | Kerim | Fırat | 489 | 1000,00 |
| 2 | 0000065127 | Cihan | Özhan | 489 | 9500,00 |

Hesaplar arası 500 birim havale işlemi başarıyla gerçekleştirildi.

## Sabitleme Noktası Oluşturmak: Save Tran

Yukarıdaki Transaction İfadelerini Anlamak bölümünde açıkladığımız sabitleme noktası işlemi için bir kaç örnek yapacağız.

Öncelikle, **SAVE TRAN** çalışma şeklini en basit haliyle kavrayabilmek için basit bir transaction oluşturalım.

```
BEGIN TRANSACTION
 SELECT * FROM Accounts;

 UPDATE Accounts
 SET Balance = 500, Branch = 287
 WHERE AccountID = '0000064219';
  SELECT * FROM Accounts;

 SAVE TRAN save_updateAccount;

 DELETE FROM Accounts WHERE AccountID = '0000064219';
  SELECT * FROM Accounts;
```

```
 ROLLBACK TRAN save_updateAccount;
  SELECT * FROM Accounts;
 ROLLBACK TRAN;
  SELECT * FROM Accounts;
COMMIT TRANSACTION;
```

Kerim Fırat isimli müşterinin hesabındaki para üzerinde güncelleme yaparak değiştiriyoruz. Aynı müşterinin şubesini de değiştirerek 287 no'lu şubeye aktarıyoruz. Daha sonra, sabitleme noktasına ve sonrasında da işlemi en başına alacak şekilde **ROLLBACK** yapıyoruz.

Sabitleme noktasına geri dönüş gerçekleştirecek bir prosedür oluşturalım. Bu prosedürde **HumanResources.JobCandidate** tablosundan veri silme işlemi gerçekleştirilmek isteniyor. Ancak iş transaction ve prosedür iş birliğine geldiğinde bir çok hata yönetimi ve **IF . . . ELSE** yapısı devreye giriyor.

```
CREATE PROCEDURE SaveTran(@InputCandidateID INT)
AS
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT;
IF @TranCounter > 0
    SAVE TRANSACTION ProcedureSave;
ELSE
BEGIN TRANSACTION;
BEGIN TRY
    DELETE HumanResources.JobCandidate
    WHERE JobCandidateID = @InputCandidateID;
    IF @TranCounter = 0
        COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  IF @TranCounter = 0
      ROLLBACK TRANSACTION;
  ELSE
      IF XACT_STATE() <> -1
          ROLLBACK TRANSACTION ProcedureSave;
  DECLARE @ErrorMessage NVARCHAR(4000);
  DECLARE @ErrorSeverity INT;
  DECLARE @ErrorState INT;
```

```
  SELECT @ErrorMessage = ERROR_MESSAGE();
  SELECT @ErrorSeverity = ERROR_SEVERITY();
  SELECT @ErrorState = ERROR_STATE();

  RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH;
```

Prosedürü çağıralım.

```
EXEC SaveTran 13;
```

# Try-Catch İle Transaction Hatası Yakalamak

SQL Server 2005'den önceki sürümlerde de kullanılabilen **@@ERROR** değeri, her seferinde kod blokları içerisinde kontrol edilerek hata durumu değerlendirilir. Bu durum programlama tarafında birçok zorluğu beraberinde getirir. SQL Server 2005 ile birlikte **Try-Catch** yapısı kullanılarak daha anlaşılır ve yönetimi kolay kod blokları oluşturulabilir.

**Söz Dizimi:**

```
BEGIN TRY
 BEGIN TRAN
 ...
 COMMIT
END TRY
BEGIN CATCH
 PRINT @@ERROR
 ROLLBACK
END CATCH;
```

İlk transaction örneğinde **@@ERROR** deyimi ile oluşturulan prosedürü **Try-Catch** yapısı ile değiştirelim.

```
ALTER PROC sp_MoneyTransfer(
 @PurchaserID CHAR(10),
 @SenderID  CHAR(10),
 @Amount  MONEY
)
```

```
AS
BEGIN TRY
 BEGIN TRANSACTION
  UPDATE Accounts
  SET Balance = Balance - @Amount
  WHERE AccountID = @SenderID

  UPDATE Accounts
  SET Balance = Balance + @Amount
  WHERE AccountID = @PurchaserID
 COMMIT
END TRY
BEGIN CATCH
 PRINT @@ERROR + ' hatası oluştuğu için havale yapılamadı.'
 ROLLBACK
END CATCH;
```

## XACT_STATE() FONKSİYONU

Transaction durumunu öğrenmek için kullanılır. Bu sistem fonksiyonunun döndürdüğü değerler ve anlamları aşağıdaki gibidir.

- `0` : Açık transaction yok.
- `-1` : Doomed (*uncommitable*) transaction var.
- `1` : Açık transaction var.

Prosedürü **`Xact_State`** fonksiyonuna göre değiştirelim.

```
ALTER PROC sp_MoneyTransfer(
 @PurchaserID CHAR(10),
 @SenderID  CHAR(10),
 @Amount  MONEY
)
AS
BEGIN TRY
 BEGIN TRANSACTION
  UPDATE Accounts
  SET Balance = Balance - @Amount
  WHERE AccountID = @SenderID
```

```
  UPDATE Accounts
  SET Balance = Balance + @Amount
  WHERE AccountID = @PurchaserID
 COMMIT
END TRY
BEGIN CATCH
IF(XACT_STATE()) = -1
BEGIN
 PRINT @@ERROR + ' kodlu hata gerçekleşti. Havale yapılamadı.'
ROLLBACK TRAN
END
IF(XACT_STATE()) = 1
BEGIN
 PRINT @ERROR + ' kodlu hata gerçekleşti. Ancak transaction başarı
ile bitirildi.'
COMMIT TRAN
END
END CATCH;
```

# İç İçe Transaction'lar (Nested Transactions)

Transaction'ların iç içe kullanılmasıyla gerçekleşen yönteme denir. Birden fazla transaction bloğunu iç içe kullanarak işlemlerin katmanlarını farklı şekillerde yönetmeye imkan verir.

Teknik olarak mümkün olsa da çok kullanılırken dikkatli olunmalıdır. Ayrıca mecbur kalınmadığı takdirde kullanılması önerilmez. Çünkü transaction zaten yapı olarak karmaşık ve hata olasılıkları olan bir işlemdir. Bu karmaşık işlemleri daha da karmaşık hale getirecek iç içe transaction kullanımı işleri daha da zorlaştırabilir.

**Söz Dizimi:**

```
BEGIN TRANSACTION   -- Dış Transaction
BEGIN TRY
 BEGIN TRAN A    -- İç Transaction
 BEGIN TRY
```

```
 -- sorgu ifadeleri
 COMMIT TRAN A
    END TRY
    BEGIN CATCH
       ROLLBACK TRAN A
    END CATCH
    ROLLBACK TRAN   -- Tüm işlemleri geri al
END TRY
BEGIN CATCH
  ROLLBACK TRAN
END CATCH
```

Normal transaction'larda olduğu gibi iç içe transaction'lar için de transaction'ın tamamlanması, yani kapatılması kaynak tüketimi ve veri tutarlılığı açısından önemlidir. İç içe transaction kullanımında bu önem daha da artmaktadır.

# Ortak Zamanlılık ve İzolasyon Seviyeleri

Bir veritabanı yönetim sistemindeki veri kaynağına birden fazla kullanıcının doğrudan erişim sağlayarak veri değiştirme işlemleri yapması mantıksal olarak basit görünse de veri uyumluluğu açısında arka plan işlemlerinde farklı bazı özellikler kullanılır. SQL Server, üzerinde işlem yapılmak istenen veri kaynağındaki veriyi diğer transaction işlemlerine karşı izole etmesi gerekir. Bunun nedeni; üzerinde değişiklik yapılmak istenen verinin bir başka kullanıcı tarafından aynı anda değiştirilme işlemine tabi tutulabilme olasılığıdır. Bir transaction, tablodaki sütun üzerinde güncelleme işlemi yaparken, bir başka transaction da aynı anda bu veriyi silmek isterse ne olur?

SQL Server transaction ile ilgili izolasyon işlemini iki türlü gerçekleştirebilir.

## Kilitleme (Locking)

Üzerinde işlem yapılan veri kaynağını başka transaction'lara karşı kilitlemek için kullanılır. Bir transaction bitimine kadar, bir satır, bir sayfa ya da bir tablonun tamamı başka transaction'ların erişimine kapatılabilir.

### Satır Versiyonlama (Row Versioning)

Satır versiyonlama ile izolasyon işleminde SQL Server her bir transaction için ilgili verinin mantıksal bir kopyasını oluşturur. Transaction, bu kopya veri üzerinde işlem gerçekleştirdiği için gerçek veri üzerinde herhangi bir kilitleme yapılmasına gerek yoktur. Her transaction ve mantıksal kopya için bir versiyonlama sistemi kullanır. Satır versiyonlama ile izolasyon yapabilmek için önceden veritabanı seviyesinde ayarlamalar yapmak gerekir. Bu izolasyon modelinde versiyonlamalar `Tempdb` sistem veritabanında saklandığı için Tempdb veritabanı için özel performans ayarlamaları yapılması gerekir.

## Ortak Zamanlı Erişim Anomalileri

Kilitleme ve versiyonlama izolasyon yöntemleri, aynı verilerin farklı transaction'lar tarafından değiştirilmesi işleminde ortaya çıkabilecek olası sorunları engellemek için kullanılır.

İzolasyon teknik olarak mümkün olsa da, bazı nedenlerle izolasyonda sızma olur ve veriler aynı anda başka transaction'lar tarafından değiştirilir ya da okunursa bazı sorunlar oluşabilir.

Ortaya çıkan bu sorunlara **Ortak Zamanlı Erişim Anomalileri** denir.

### Kayıp Güncelleme (Lost Update)

Aynı kayıt üzerinde aynı anda birçok transaction işleminin değişiklik yapması durumunda son yapılan değiştirme işleminin dediği olur. Bu durumda son değişiklikten öncekiler kayıptır. Basit gibi görünse de önemli bir takım hataya yol açar.

Sinema için bir bilet almak için online satın alma işlemi gerçekleştirdiniz. İşleminiz bir transaction ile gerçekleştirilmeye başlanır. Aynı anda bir başka kişi de satın alma işlemi gerçekleştirdi ve onun için de bir transaction başlatıldı. Bu işlemler ortak zamanlı gerçekleştiği ve veriler üzerinde bir kilit ya da satır versiyonlama bulunmadığı için iki transaction da tamamlanır. Ancak son tamamlanan transaction işleminin dediği olur. Bu durumda bir bileti iki ya da daha fazla kişi satın almış olabilir. Basit görünen büyük bir hata olduğunu daha net anlamış olmalısınız.

### Tekrarlanamayan Okuma (Non-Repeatable Read)

Bir transaction aynı anda bir başka transaction'ın değiştirdiği veriler üzerinde çalışabilir. Bu nedenle transaction ilk açıldıktan sonra okuduğu veri kümesini tekrar okuyamaz. Bu sorunu engellemek için kayıtları kilitleme yöntemi kullanılmalıdır.

### Hayalet Okuma (Fantom Read)

Bir transaction, değişikliklere karşı kilitleme yaptığı için ilk okuyabildiği verileri daha sonra tekrar okuyabiliyordur. Ama yeni veri eklenmesine engel olamıyordur. Bu durumda başlayan transaction'dan sonra eklenen kayıtların veri tutarlılığını riske atmaması için veri ekleme işlemi de kilitlenmelidir.

### Kirli Okuma (Dirty Read)

Aynı kaynağa erişen birden fazla transaction'dan biri, değiştirme işlemini sona erdirmeden önce, diğer transaction'lar tarafından okunan kayıtlar gerçek veriler değildir. Bu veriler, henüz tamamlanmamış bir transaction tarafından değiştirilmiştir. Bir transaction, veri üzerinde değişiklik yaparsa ve bu sırada başka bir transaction da bu verileri okuduktan sonra, değişikliği yapan ilk transaction, yaptığı değişiklikleri geri alırsa, sonraki transaction'ların elde ettiği veriler gerçek veriler olmazlar. Bu sorunu önlemek için, ikinci transaction'ın ilk transaction bitene kadar kirli sayfaları okumasını engellemek gerekir.

## Kilitler

SQL Server'da, aynı veri üzerinde, aynı anda, birden fazla kullanıcı tarafından oluşturulacak ifade ve transaction'lar bazı durumlarda işlem çakışmasına sebep olabilir. Bir veri kaynağı üzerinde, bir transaction tarafından değiştirme işlemi gerçekleşirken, bir başka transaction tarafından aynı veri kaynağı üzerinde farklı bir işlem gerçekleştirilmek istenebilir. Bir transaction tamamlanmadan veri tutarlılığı gerçekleşmeyeceği için veri kaynağını transaction tamamlanana kadar diğer ifade ve transaction'lara erişilemez hale getirmek gerekir. Bu engelleme işlemi kilitler tarafından gerçekleştirilir.

### Kilitlenebilir Kaynaklar

- **Veritabanı:** Veritabanı tamamen kilitlenir. Bu genellikle veritabanında her şeyin değişmesini sağlayan şema (*schema*) değişikliklerinde meydana gelir.
- **Tablo:** Tablo tamamen kilitlenir. Bu kilit, tablo ile ilgili gerçek veri ve tüm nesneleri içerir.
- **Extent:** Tüm extent kilitlenir. Bir extent, sekiz page'den meydana gelir. Bu sekiz page, üzerindeki indeks page'leri ile veri satırları da etkilenir.
- **Page:** Page üzerindeki tüm veri ve indeks key'leri kilitlenir.
- **Key:** Bir indeksteki key ya da key'ler üzerinde kilitleme yapılır.
- **Satır/RID:** Kilit Satır Tanımlayıcı (***R**ow **Id**entifier, **RID***) üzerinde kilitleme yaparak tüm satırı etkiler.

## Kilit Modları

Kaynakları kilitleme ihtiyacının bir çok çeşidi olduğu gibi, kilitleme modlarının da bir çok çeşidi vardır. Bazı metodlar birbiriyle uyumsuz iken bazıları hiyerarşik olarak alt ya da üst seviyede tanımlanarak uyumluluk gösterebilir.

Kilit modlarını inceleyelim.

### Paylaşılmış Kilit (Shared Lock)

Shared Lock (*paylaşılmış kilit*), herhangi bir değişiklik yapmadan, sadece veriyi okumanız gerektiğinde kullanılır ve en basit kilit tipidir. Kullanıcıları Dirty Read problemlerinden korur.

### Özel Kilit (Exclusive Lock)

Başka bir kilit varsa Exclusive Lock oluşturulamaz. Ya da Exclusive Lock varsa o nesne üzerinde başka bir kilit oluşturulamaz. Yani, kilitli veri üzerinde aynı anda iki değiştirme(güncelleme, silme vb.) işlemi yapılamaz.

### Güncelleştirme Kilidi (Update Lock)

Update Lock, özel bir yer tutucudur. Gerçekten hangi verinin güncellenmesi gerektiğini belirlemek için, veriyi taradıktan sonra Shared Lock'un özel kilit haline gelmesi anlamına gelir. Update Lock, kilitlenmeye engel olur. Engel olunan şey bir kilit değil, çıkmaza girme anlamındaki bir kilitlenmedir.

Update Lock'lar; sadece Shared Lock ve Intent Shared Lock'lar ile uyumludur.

## Amaç Kilidi (Intent Lock)

Veri ve kilitler arasındaki nesne hiyerarşisini çözmek için kullanılır. Örneğin; tablo üzerinde oluşturulan bir kilit varsa tüm tabloyu kilitleyecektir. Satır üzerinde oluşturulan kilit ise satırı kilitler. Bir satır üzerinde kilide sahip olduğunuzda, bir başkası tablo seviyeli kilit oluşturursa tablodaki satır üzerinde ne tür bir işlem yapılmalıdır? Bu tür durumlarda SQL Server tablo içerisindeki tüm satırlarda kilit aramak yerine tabloda kilit olup olmadığına bakar.

Böyle bir kilitlemeden dolayı sorun yaşamamak için Intent Lock'lar kullanılır. Intent Lock'lar kendi arasında üçe ayrılır.

- **Paylaşılmış Amaç Kilidi (*Intent Shared Lock*):** Bu kilit türü ile shared lock hiyerarşisi düşük seviyede kurulur. Düşük seviyeden kasıt page ya da tablo gibi alt katman üzerinde kurulmasıdır. Yani, bir page üzerinde kurulabilir.
- **Özel Amaç Kilidi (*Intent Exclusive Lock*):** Intent Shared Lock ile aynıdır. Tek farkı, düşük seviye nesne üzerinde özel kilit olmasıdır.
- **Özel Amaç Kilidi İle Paylaşım (*Shared With Intent Exclusive Lock*):** Shared Lock ve Intent Exclusive Lock özelliklerini içerir. Shared Lock, nesne hiyerarşisini düşürmek, Intent Lock ise veriyi değiştirmek amacıyla kurulmuştur.

## Şema Kilitleri (Schema Locks)

Şemalar ile ilgili kilitlerdir ve iki çeşittir.

- **Şema Değişiklik Kilidi (*Schema Modification Lock*):** Nesneye şema değişikliği yapılır. Kilit süresi içinde bu nesne üzerinde `CREATE`, `ALTER` ve `DROP` ifadeleri çalıştırılabilir.
- **Şema Denge Kilidi (*Schema Stability Lock*):** Bu kilidin amacı, nesne üzerinde diğer sorguların aktif kilitleri olduğu için Schema Modification Lock kilidi kurulmasını engellemektir.
- **Bulk Updates Lock:** Verinin paralel yüklenmesine izin verir. Yani, tablo diğer eylemler tarafından kilitlense de birden fazla `BULK INSERT` ya da `BCP` işlemi gerçekleştirilebilir.

# Optimizer İpuçları ile Özel Bir Kilit Tipi Belirlemek

SQL Server'da kilitleme işlemlerindeki kontrolü ele almak için kullanılır. SQL ifadelerinde tablodan sonra kullanılırlar.

**Söz Dizimi:**

```
FROM tablo_ismi [AS takma_isim] [[WITH] (ipucu)]
```

**Kullanım Yöntemleri:**

```
FROM tablo_ismi AS takma_isim WITH (ipucu)
FROM tablo_ismi AS takma_isim (ipucu)
FROM tablo_ismi WITH (ipucu)
FROM tablo_ismi (ipucu)
```

Bu ipuçları, SQL Server kilit yöneticisine bırakmaksızın ifadeler için kilit tipi belirleyebilmeyi sağlar. yukarıdaki kullanım yöntemlerinden herhangi biriyle kullanılabilir.

## READCOMMITTED

SQL Server için varsayılan özelliktir. **READ_COMMITTED_SNAPSHOT** özelliği aktifse bu kilit uygulanmaz.

**Örnek:**

```
SELECT * FROM Production.Product (READCOMMITTED);
```

## READUNCOMMITTED/NOLOCK

Hiç bir kilit elde edilmez. Dirty Read problemleri dahil bir çok probleme neden olabilir.

**Örnek:**

```
SELECT * FROM Production.Product (READUNCOMMITTED);
```

ya da

```
SELECT * FROM Production.Product (NOLOCK);
```

## READCOMMITTEDLOCK

**READCOMMITTED** ile aynı işi yapar. Tek fark, bir kilit nesne için daha fazla gerekmediğinde bırakılır.

**Örnek:**

```
SELECT * FROM Production.Product (READCOMMITTEDLOCK);
```

## SERIALIZABLE/HOLDLOCK

Yüksek izolasyon seviyedir ve veri uyumluluğunu garanti eder. Transaction işleminde kilit kurulduktan sonra **ROLLBACK** ya da **COMMIT** ile transaction sonlanana kadar serbest bırakılmaz.

**Örnek:**

```
SELECT * FROM Production.Product (SERIALIZABLE);
```

ya da

```
SELECT * FROM Production.Product (HOLDLOCK);
```

## REPEATABLEREAD

Transaction'da kilit kurulduktan sonra **ROLLBACK** ya da **COMMIT** ile transaction sonlanana kadar serbest bırakılmaz. Ancak, yeni veri eklenebilir.

**Örnek:**

```
SELECT * FROM Production.Product (REPEATABLEREAD);
```

## READPAST

Kilidin serbest bırakılmasını beklemek yerine **Page**, **Extent** ve tablo kilitleri hariç tüm satır kilitleri atlanır.

**Örnek:**

```
SELECT * FROM Production.Product (READPAST);
```

## ROWLOCK

Kilidin başlangıç seviyesini satır seviyesinde olmaya zorlar. Kilit sayısı sistemin kilit eşik değerini aşmışsa, kilidin daha düşük granular seviyelerine yükseltilmesi engellenmez.

## PAGLOCK

Adından da anlaşılacağı gibi Page seviyeli kilit kullanır.

**Örnek:**

```
SELECT * FROM Production.Product (PAGLOCK);
```

## TABLOCK

Tablo kilit uygulanmasını sağlar. Tablo taraması durumlarını hızlandırır.

**Örnek:**

```
SELECT * FROM Production.Product (TABLOCK);
```

## TABLOCKX

Adından da anlaşılacağı gibi **TABLOCK** özelliğine benzer şekilde çalışır. İzolasyon seviyesi ayarlamasına bağlı olarak transaction ya da ifade süresince kullanıcıların tabloya erişmesine izin vermez, tabloyu kilitler.

**Örnek:**

```
SELECT * FROM Production.Product (TABLOCKX);
```

## UPDLOCK

Adından da anlaşılacağı gibi Update Lock kullanır. Diğer kullanıcıların Shared Lock'ları elde etmesine izin verir, ancak ifade ya da transaction sonlanana kadar veri değişikliğine izin vermez.

**Örnek:**

```
SELECT * FROM Production.Product (UPDLOCK);
```

### XLOCK

Kilit granularity seviyesinin nasıl belirlendiğine bakmaksızın özel kilit belirleyebilmeyi sağlar.

**Örnek:**

```
SELECT * FROM Production.Product (XLOCK);
```

Bu ipuçları, sadece tek satırlık ve tek tabloluk ifadelerde değil, aynı zamanda **JOIN** işlemlerinde bir tablo üzerinde kurulmak için de kullanılabilir.

**Örnek:**

```
SELECT * FROM tablo_ismi1 (ipucu)
JOIN tablo_ismi2 AS ti2
ON tablo_ismi1.sutunID1 = ti2.sutunID1;
```

## Isolation Seviyesinin Ayarlanması

Transaction işlemlerinde, aynı anda veri kaynaklarına erişen birden fazla transaction'ın bir diğerinin kaynak erişimlerinden ve değiştirdiği verilerden izole edilmesi ile ilgili ayarlardır.

### READ COMMITTED

SQL Server'da varsayılan seçenektir. Veri okurken kirli okumayı önler. Ancak transaction bitmeden başka bir transaction tarafından veri değişikliği yapılabilir.

**READ COMMITTED** ile varsayılan bu seviyenin korunması yoluyla **Dirty Read** problemlerini önleyecek yeterli doğruluğa sahip olunduğundan emin olunabilir. Bununla birlikte, **Non-Repeatable Read** ve **Phantom** problemleri oluşabilir.

## READ UNCOMMITTED

İzolasyon seviyesi 0 (sıfır), yani hiç bir izolasyon yoktur. Transaction açıkken başka transaction'da veriler üzerinde değişiklik yapabilir. İzolasyon seviyesini **READ UNCOMMITTED** olarak ayarlamak, SQL Server'a herhangi bir kilit oluşturmamasını söyler. Bu izolasyon seviyesi ile çoğu Dirty Read olmak üzere bir çok uyumluluk sorunu ile karşılaşılabilir.

**READ UNCOMMITTED** veri doğruluğu açısından uygun bir izolasyon seçimi değildir. Kullanım amacı da bu tür durumlardır. Bir raporlama işlemi bazen epeyce uzun sürebilir. Bu süre içerisinde tablo ya da satırlarda kilitler oluşturacak izolasyon seviyeleri kullanmak diğer transaction'lar tarafından verilerin okunamaması ya da değiştirilememesi anlamına gelir. Rapor işlemlerinde, genellikle veriler için yüksek veri tutarlılığa ihtiyaç yoktur. Yaklaşık ve orantısal bir işlem için, yeteri kadar tutarlı veri kullanılması gerekir. Uzun sürecek raporlama işlemlerinde bu izolasyon seviyesinin kullanılması önerilebilir.

## REPEATABLE READ

Transaction içerisindeki sorguda geçen tüm veriler kilitlemeye alınır. Kirli hafızanın okunmasına izin vermez.

İzolasyon seviyesini yükselten **REPEATABLE READ** ile sadece Dirty Read değil, Non-Repeatable Read durumları da engelleyerek ek seviyede uyumluluk koruması sağlar.

## SERIALIZABLE

Bir transaction sonlanmadan, aynı anda başka bir transaction ortak kaynaklara erişemez. Lost Update dışında tüm uyumluluk problemlerini engeller. En katı izolasyon seviyesidir.

Bu izolasyon seviyesinde, kullanıcı transaction ile ilgi bir işlem yapıyorsa, başka bir transaction işlemi gerçekleştirmek için bu transaction'ın tamamlanmasını beklemek zorundadır.

## SNAPSHOT

Bir transaction başladığında veritabanının o anki kopyasını alır. Snapshot üzerinde satır versiyonlama yapılabilir. Her ifade çalıştırılırken, o anki gerçek veri alındıktan sonra üzerinde işlem yapılması sağlanabilir.

# İzolasyon Seviyesi Yönetimi

SQL Server'da izolasyon işlemleri için aşağıdaki söz dizimi kullanılır.

**Söz Dizimi:**

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED
| REPEATABLE READ | SNAPSHOT | SERIALIZABLE }
```

SQL Server, izolasyon seviyelerini oturum bazında ayarlamak için destek sağlar.

Oturum ile ilgili tüm parametrelerin listesini almak için;

```
DBCC USEROPTIONS;
```

Yukarıdaki sorgu sonucundaki izolasyon seviyesi ile ilgili olan kısım;

| | Set Option | Value |
|---|---|---|
| 1 | textsize | 2147483647 |
| 2 | language | us_english |
| 3 | dateformat | mdy |
| 4 | datefirst | 7 |
| 5 | lock_timeout | -1 |
| 6 | quoted_identifier | SET |
| 7 | arithabort | SET |
| 8 | ansi_null_dflt_on | SET |
| 9 | ansi_warnings | SET |
| 10 | ansi_padding | SET |
| 11 | ansi_nulls | SET |
| 12 | concat_null_yi... | SET |
| 13 | isolation level | read committed |

Oturumdaki izolasyon seviyesini değiştirmek için aşağıdaki ifade kullanılabilir.

**Söz Dizimi:**

```
SET TRANSACTION ISOLATION LEVEL izolasyon_seviyesi
```

İzolasyon seviyesini anlayabilmek için bir güncelleme işlemi gerçekleştirelim.

Ürün fiyatlarında %7'lik bir zam yapacağız.

Öncelikle kayıtların ilk halini görelim.

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

| | Name | ListPrice |
|---|---|---|
| 1 | Road-150 Red, 62 | 3828,7489 |
| 2 | Road-150 Red, 44 | 3828,7489 |
| 3 | Road-150 Red, 48 | 3828,7489 |
| 4 | Road-150 Red, 52 | 3828,7489 |
| 5 | Road-150 Red, 56 | 3828,7489 |
| 6 | Mountain-100 Silver, 38 | 3637,9893 |
| 7 | Mountain-100 Silver, 42 | 3637,9893 |

Daha sonra, güncelleme yapmak için bir transaction oluşturalım.

```
BEGIN TRAN
 UPDATE Production.Product
 SET ListPrice =  ListPrice * 1.07
```

Transaction açıldı ve güncelleme işlemi gerçekleşti. Yeni bir sorgu ekranı açalım ve güncellenen veriyi kontrol etmek için veri listeleme sorgusunu bu yeni sorgu ekranında yazarak çalıştıralım.

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

Bu **SELECT** sorgusu sonucunu göremeyeceksiniz. Çünkü işlem sonlanmayacaktır. Bunun nedeni, SQL Server varsayılan izolasyon seviyesidir. Bu izolasyon seviyesi hatırlarsanız kirli hafızaya erişmenize engel olacak şekilde tasarlanmıştı.

İzolasyon seviyesini değiştirerek aynı işlemi gerçekleştirebiliriz. Oturum kirli hafızadan okumaya izin veren **READ UNCOMMITTED** izolasyon seviyesine göre ayarlandığında değişen veri listelenebilir.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT Name, ListPrice FROM Production.Product ORDER BY ListPrice DESC;
```

| | Name | ListPrice |
|---|---|---|
| 1 | Road-150 Red, 62 | 3828,7489 |
| 2 | Road-150 Red, 44 | 3828,7489 |
| 3 | Road-150 Red, 48 | 3828,7489 |
| 4 | Road-150 Red, 52 | 3828,7489 |
| 5 | Road-150 Red, 56 | 3828,7489 |
| 6 | Mountain-100 Silver, 38 | 3637,9893 |
| 7 | Mountain-100 Silver, 42 | 3637,9893 |

Aynı işlem, transaction seviyeli gerçekleştirmek yerine tek sorgu için de oluşturulabilir.

```
SELECT Name, ListPrice FROM Production.Product (NOLOCK)
ORDER BY ListPrice DESC;
```

| | Name | ListPrice |
|---|---|---|
| 1 | Road-150 Red, 62 | 3578,27 |
| 2 | Road-150 Red, 44 | 3578,27 |
| 3 | Road-150 Red, 48 | 3578,27 |
| 4 | Road-150 Red, 52 | 3578,27 |
| 5 | Road-150 Red, 56 | 3578,27 |
| 6 | Mountain-100 Silver, 38 | 3399,99 |
| 7 | Mountain-100 Silver, 42 | 3399,99 |

Sonuçlar istendiği gibi kilitten bağımsız olarak listelendi. Oluşturduğumuz güncelleme işlemini geri alarak tekrar veri listelemesi gerçekleştirelim. **ROLLBACK** komutunu **BEGIN TRANSACTION** ile transaction'ı başlattığınız sorgu ekranında çalıştırmalısınız.

```
ROLLBACK
```

Güncelleme yapılarak kirli hafızada bulunan değişikliklerin gerçek veriye etki etmeden geri alındığını ve önceki verinin gösterildiği görülecektir.

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

| | ProductID | Name |
|---|---|---|
| 1 | 1 | Adjustable Race |
| 2 | 879 | All-Purpose Bike Stand |
| 3 | 712 | AWC Logo Cap |
| 4 | 3 | BB Ball Bearing |
| 5 | 2 | Bearing Ball |
| 6 | 877 | Bike Wash - Dissolver |
| 7 | 316 | Blade |

**READ COMMITTED** izolasyon seviyesine geçmek için;

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

**SERIALIZABLE** izolasyon seviyesine geçmek için;

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## Transaction Bazlı Snapshot İzolasyon

Transaction işleminde verinin bir kopyasının oluşturulması sağlanabilir. Transaction'ın snapshot'ı, yani kopyası alınır ve transaction kapanıncaya kadar saklanır. Verilerin az değişikliğe (**Update**) uğradığı, ama çok görüntüleme (**Select**) durumlarda kullanılabilir.

Snapshot izolasyon seviyesini kullanmak için veritabanı bazında aşağıdaki ayar yapılmalıdır.

```
USE MASTER
ALTER DATABASE AdventureWorks
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Gerekli ayarlamalardan sonra Snapshot izolasyon ile bir transaction gerçekleştirmek için şu genel ifadeler kullanılabilir.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRAN
 SELECT ProductID, Name FROM Production.Product;
COMMIT TRAN
```

| | ProductID | Name |
|---|---|---|
| 1 | 1 | Adjustable Race |
| 2 | 879 | All-Purpose Bike Stand |
| 3 | 712 | AWC Logo Cap |
| 4 | 3 | BB Ball Bearing |
| 5 | 2 | Bearing Ball |
| 6 | 877 | Bike Wash - Dissolver |
| 7 | 316 | Blade |

### İfade Bazlı Snapshot İzolasyon

İfade bazlı snapshot izolasyon, çalıştırılan her ifade için ayrı kopya (*snapshot*) alır. Bu izolasyon yöntemi ile, devam eden transaction içerisinde, her ifade için yeni bir snapshot almasından dolayı verinin son halini transaction'da kullanmayı sağlar.

İfade bazlı snapshot izolasyonu kullanabilmek için;

```
USE MASTER
ALTER DATABASE AdventureWorks
SET READ_COMMITTED_SNAPSHOT ON
```

Yapılan bu değişiklik ile birlikte, artık **READ COMMITTED** izolasyon seviyesi ile başlatılan transaction'lar için versiyonlama yapmaya başlayacaktır.

## Kilitlenmeleri Yönetmek

SQL Server'da izolasyon gerçekleştirirken kilitleme seçeneği büyük öneme sahiptir. Kilitlerin yönetimi, çalışma prensipleri, veritabanındaki kilitli nesneler, kilitlerin takip edilmesi ve görüntülenmesi, sistem tarafındaki kilit mekanizmalarının incelenmesi gibi konuları bu bölümde ele alacağız.

## Kilitlemeleri Gözlemlemek

Sistemdeki kilitlemeleri kimlerin yaptığını takip etmek için **sp_who** sistem prosedürü kullanılır.

```
sp_who ['kulanici_ismi']
```

Sistemdeki tüm kullanıcılar için kilitleme bilgilerini listeleyelim.

```
sp_who
```

| | spid | ecid | status | loginame | hostname | blk | dbname | cmd | request_id |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | background | sa | | 0 | NULL | LOG WRITER | 0 |
| 2 | 2 | 0 | background | sa | | 0 | NULL | RECOVERY WRITER | 0 |
| 3 | 3 | 0 | background | sa | | 0 | NULL | LAZY WRITER | 0 |
| 4 | 4 | 0 | background | sa | | 0 | master | SIGNAL HANDLER | 0 |
| 5 | 5 | 0 | background | sa | | 0 | NULL | LOCK MONITOR | 0 |
| 6 | 6 | 0 | background | sa | | 0 | NULL | XE DISPATCHER | 0 |
| 7 | 7 | 0 | background | sa | | 0 | NULL | XE TIMER | 0 |

Parametresiz kullanılabilen bu sistem prosedürü ile tüm kilitlemeler listelenebileceği gibi bir kullanıcı ismi verilerek, sadece o kullanıcının kilitlemeleri de listelenebilir.

```
sp_who 'sa'
```

| | spid | ecid | status | loginame | hostname | blk | dbname | cmd | request_id |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | background | sa | | 0 | NULL | LOG WRITER | 0 |
| 2 | 2 | 0 | background | sa | | 0 | NULL | RECOVERY WRITER | 0 |
| 3 | 3 | 0 | background | sa | | 0 | NULL | LAZY WRITER | 0 |
| 4 | 4 | 0 | background | sa | | 0 | master | SIGNAL HANDLER | 0 |
| 5 | 5 | 0 | background | sa | | 0 | NULL | LOCK MONITOR | 0 |
| 6 | 6 | 0 | background | sa | | 0 | NULL | XE DISPATCHER | 0 |
| 7 | 7 | 0 | background | sa | | 0 | NULL | XE TIMER | 0 |

**sp_who** sistem prosedürü ile tüm kullanıcılar için hangi kullanıcının hangi kaynaklara eriştiğini ve durumlarını görebiliriz.

**sp_who** sistem prosedürünün çalışması sonucu 9 sütunluk bir kayıt listelenir. Bu sütunların isimleri ve anlamları sırası ile şöyledir.

- **spid:** Sistem işlem (*process*) **ID** değeri.
- **ecid:** Çalıştırma içerik değeri.
- **status:** İşlemin durumu.
- **loginame:** İşlemin sahibi olan kullanıcının adı.
- **hostname:** İşlemin sahibi bilgisayarın adı.
- **blk:** İşlemi kilitleyen bir başka işlem varsa o işlemin **ID** değeri. Eğer yok ise 0'dır.
- **dbname:** İşlemin kullandığı veritabanının adı.
- **cmd:** Çalıştırılan SQL ifadesi ya da SQL Server motoru işlem adı.
- **request_id:** Özel nedenlerden dolayı çalışan işlemlerin istek numaralarını verir. Yok ise, değer 0'dır.

**sp_who** gibi **sp_who2** sistem prosedürü de kullanılabilir.

| | SPID | Status | Login | HostName | BlkBy | DBName | Command | CPUTime | DiskIO | LastBatch | ProgramName | SPID | REQUESTID |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | BACKGROUND | sa | . | . | NULL | LOG WRITER | 15 | 0 | 01/30 11:27:27 | | 1 | 0 |
| 2 | 2 | BACKGROUND | sa | . | . | NULL | RECOVERY WRITER | 15 | 0 | 01/30 11:27:27 | | 2 | 0 |
| 3 | 3 | BACKGROUND | sa | . | . | NULL | LAZY WRITER | 390 | 0 | 01/30 11:27:27 | | 3 | 0 |
| 4 | 4 | BACKGROUND | sa | . | . | master | SIGNAL HANDLER | 0 | 0 | 01/30 11:27:27 | | 4 | 0 |
| 5 | 5 | BACKGROUND | sa | . | . | NULL | LOCK MONITOR | 0 | 0 | 01/30 11:27:27 | | 5 | 0 |
| 6 | 6 | BACKGROUND | sa | . | . | NULL | XE DISPATCHER | 0 | 0 | 01/30 11:27:27 | | 6 | 0 |
| 7 | 7 | BACKGROUND | sa | . | . | NULL | XE TIMER | 46 | 0 | 01/30 11:27:27 | | 7 | 0 |

**sp_lock** sistem prosedürü kilitli olan kaynakları görüntülemek için kullanılır. Hangi kaynakların, hangi kullanıcılar tarafından kilitlendiği görülebilir.

**Söz Dizimi:**

```
sp_lock [@spid1 = 'spid1'][,@spid2='spid2']
```

**sp_lock** sistem prosedürü dışarıdan iki parametre alabiliyor olsa da, parametresiz olarak en basit haliyle kullanılabilir.

```
sp_lock
```

| | spid | dbid | ObjId | IndId | Type | Resource | Mode | Status |
|---|---|---|---|---|---|---|---|---|
| 1 | 52 | 7 | 0 | 0 | DB | | S | GRANT |
| 2 | 53 | 5 | 0 | 0 | DB | | S | GRANT |
| 3 | 55 | 2 | 5 | 0 | TAB | | IX | GRANT |
| 4 | 55 | 2 | 7 | 0 | TAB | | IX | GRANT |
| 5 | 55 | 7 | 0 | 0 | DB | | S | GRANT |
| 6 | 55 | 2 | 3 | 0 | TAB | | IX | GRANT |
| 7 | 55 | 2 | 34 | 0 | TAB | | IX | GRANT |

Parametreli kullanımı da kolaydır.

- **Tek parametreli `sp_lock`:** Bir işlem `ID` (**`processesID`**) değeri alır. `ID` değerini aldığı işlem tarafından gerçekleştirilen kilitlenmeleri gösterir.
- **Çift parametreli `sp_lock`:** İki işlem `ID` (**`processesID`**) değeri alır. Aldığı iki `ID` değeri tarafından ortak gerçekleştirilen kilitlenmeleri gösterir.

**sp_lock** sonucunda listelenen sütunların anlamları;

- **`spid`:** Sistem işlem (*process*) `ID` değeri.
- **`dbid`:** Kilitlenme isteyen veritabanının sistem `ID`'si.
- **`ObjId`:** Kilitlemede bulunmak isteyen nesnenin sistem `ID`'si.
- **`IndId`:** İndeks `ID`'si.
- **`Type`:** Kilitlemenin tipi.
- **`Resource`:** Kilitlenen kaynak.
- **`Mode`:** Kilitleme isteyenin kilitleme modu. (**`Shared`**, **`Exclusive`** vb.).
- **`Status`:** Kilitleme istek durumu (**`Granted`**, **`Convert`**, **`Waiting`** vb.).

Uygulamalarda **sp_lock** kullanmanız tavsiye edilmez. Microsoft, SQL Server'ın ilerleyen sürümlerinde **sp_lock** yerine, **sys.dm_tran_locks** sistem katalog view'i kullanmayı planlamaktadır.

## Zaman Aşımını Ayarlamak

SQL Server'da fark ettiyseniz transaction'ların çalışma süresi sınırsızdır. **BEGIN TRAN** ile başlattığınız transaction, siz **COMMIT** ya da **ROLLBACK** yapana kadar devam edecektir. Bu durum sistem kaynaklarının sürekli tükenmesi anlamına gelir.

Transaction'ların kaynaklar üzerinde bir süreliğine kilitleme yapabilmelerini sağlamak için **LOCK_TIMEOUT** oturum parametresi kullanılır. Parametre olarak milisaniye alan bu oturum parametresi ile transaction zaman aşımı süresini ve dolayısıyla sistem kaynaklarının performans açısından düzenlenmesini sağlayabilirsiniz.

Varsayılan olarak sınırsız olan zaman aşımı süresinin SQL Server tarafındaki ayar karşılığı **-1**'dir.

```
SELECT @@LOCK_TIMEOUT
```

| | (No column name) |
|---|---|
| 1 | -1 |

Bu ayarı değiştirmek için ise, **LOCK_TIMEOUT** oturum parametresini **SET** etmek gerekir.

Zaman aşımı süresini on bin milisaniye yapalım.

```
SET LOCK_TIMEOUT 10000
```

Parametre değerine tekrar bakalım.

```
SELECT @@LOCK_TIMEOUT
```

| | (No column name) |
|---|---|
| 1 | 10000 |

Zaman aşımı süresini tekrar sınırsız yapmak için **-1** değeri ile **SET** etmeniz yeterlidir.

```
SET LOCK_TIMEOUT -1
```

**LOCK_TIMEOUT** değerini tekrar kontrol edelim.

```
SELECT @@LOCK_TIMEOUT    15.PNG
```

| | (No column name) |
|---|---|
| 1 | -1 |

# Kilitleme Çıkmazı: Deadlock

Deadlock (*ölüm kilitlenmesi*), kullanıcıların işlem yaparken, karşılıklı olarak birbirlerinin bir sonraki işlemlerini engelleyen durumlar için kullanılan terimdir. SQL Server böyle bir durumu fark ettiğinde 1205 numaralı mesaj kodunu kullanılır. SQL Server böyle bir deadlock durumunda rastlantısal olarak bir kurban seçer ve transaction'ı **ROLLBACK** ile iptal ederek tüm işlemleri geri alır.

Bir geliştirici olarak bu rastlantısallığa müdahale ederek kurban seçiminde söz sahibi olunabilir.

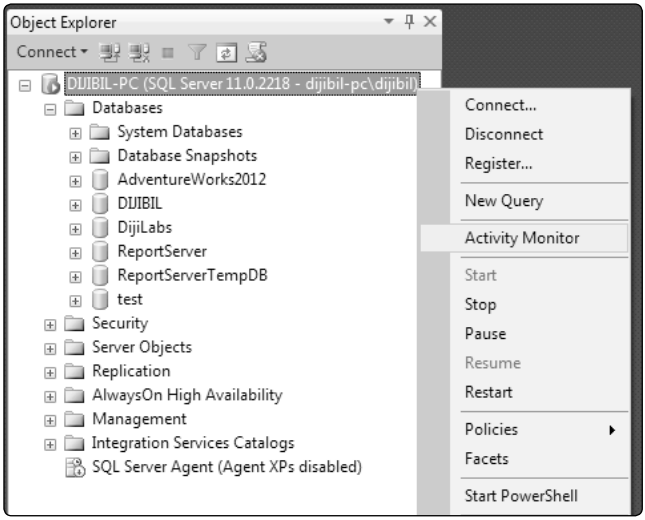
Bu işlem için **DEADLOCK_PRIORITY** parametresi kullanılır.

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority>
@deadlock_var | @deadlock_intvar }
<numeric-priority> ::= { -10|-9|-8|…|0|…|8|9|10 }
```

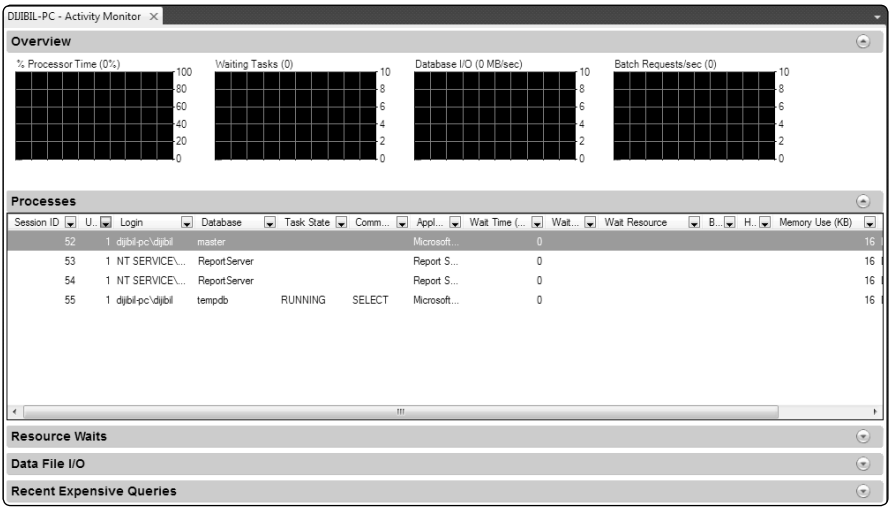
**LOW**, **NORMAL** ya da **HIGH** yerine **-10** ile **10** arasında bir değer de atanabilir.

## Aktivite Monitörü ile Kilitlenmeleri Takip Etmek ve Process Öldürmek

SSMS ile kilitlenmeleri takip etmek için, öncelikle **Aktivite Monitör** açalım.



Resimdeki menüye girdiğinizde aşağıdaki gibi bir ekran ile karşılaşacaksınız.




Ekranda parametreleri inceledikten sonra aşağıdaki sorguları çalıştıralım.

Bu işlemler için iki sorgu ekranı kullanacağız. İlk sorgu ekranında;

```
BEGIN TRAN
UPDATE Accounts
SET Balance = Balance - 500
WHERE AccountID = '0000065127';
```

İkinci sorgu ekranında;

```
SELECT * FROM Accounts;
```

Sorgular çalıştırıldıktan sonra aktivite monitörüne tekrar bakıldığında, bir kilitlenme gerçekleştiği görülecektir.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 56 | 1 | dijibil-pc\dijibil | DIJIBIL | SUSPENDED | SELECT | Microsoft... | 34941 | LCK_M_S | keylock hobtid=7205... | 51 | 16 |

Şuan kilitlenmiş bir sorgu ekranındaysanız, yeni bir sorgu ekranı oluşturun ve aşağıdaki **işlem öldürme** (*Process Kill*) komutunu kullanarak belirlediğiniz işlem `ID` değerindeki işlemi sonlandırın.

```
KILL 56
```