**Dynamic SQL**

Çalıştırma anına kadar yazılacak sorgunun tam olarak ne olduğunu bilmediğimiz bazı durumlarla karşılaşabiliriz. Bu durumlarda SQL Server bu tür durumlar için iki seçenek sunar. **EXEC komutu veya sp\_executesql** prosedürü kullanılır. Peki bunlardan hangisini ne zaman tercih etmeliyiz. Normal şartlar altında bir SQL cümlesini doğrudan çalıştırdığımız SQL Server, bellekten ilgili yürütme planını okuyup ona göre sorguyu çalıştırır. Böylece daha önceki deneyimleri doğrultusunda en uygun okumayı tercih etmiş olur. Fakat dinamik cümleleri çalıştırırken execution plan kullanılmaz her çalıştırmada yeni bir execution plan oluşturulur. Bu iki komutun en önemli farkı **sp\_executesql**, parametrik olabilirken EXEC / EXECUTE parametrik değildir. Buradaki parametreden kastımız bu komutlara parametre olarak verilmiş sorgunun parametrik olmasıdır. Asıl önemli olan konu performans açısından sunucu aynı execution plan’ı kullandığı için daha sp\_executesql yordamı daha iyi sonuç vermektedir. Çünkü bu yordamın mantığı sorgunun aynı olduğu sadece parametrenin farklı çalıştırılıyor olmasıdır.

**1.Dinamik SQL Oluşturma (Execute())**

Basit ama dinamik, dışarıdan karakter olarak dinamik SQL ifadesi alan sorgularımız **Exec[ute]()** fonksiyonunu kullanırız. Genel kullanımı aşağıdaki gibidir.

EXECUTE ({ @string\_degisken | [N] 'T-SQL ifadesi' } [ + ....n ])

Örneğin tablo adını bilmeden SELECT yazmamız gerekirse, tablo adını daha sonra verebileceğimiz şekilde bir SELECT ifaedesi aşağıdaki gibi olabilir:

declare @sorgu varchar(3000), @tablo varchar(20)

set @tablo='Customers'

set @sorgu= 'Select \* from ' + @tablo

Exec(@sorgu)

Exec() fonksiyonunda tanımlanan değişkene dıaşarıdan erişilemez. Exec() içerisinden de dışarıdaki bir değişken doğrudan adı ile çağırılamaz ama değeri önceden alınabilir. Örneğin, bir store procedure yapıp parametre olarak belirtelim:

create proc Sorgular (

@sorgu varchar(3000)

)

as

exec(@sorgu)

--Store Procedure u çalıştırmak için

exec Sorgular 'select \* from Customers'

**Not:** Exec() foksiyonu ile çalıştırılan dinamik sorgular için çalıştırma planı tutulmaz. Bu yüzden doğrudan çalıştırılabilecek bir şeyi Exec() fonksiyonu içerisinde çalıştırmak veritabanı performansınızı olumuz etkileyebilir.

Örneğin London da bulunan ve müşterilerin ID si ContactName i ve City bilgilerini listeleyelim:

DECLARE @columnList varchar(75)

DECLARE @city varchar(75)

SET @columnList = 'CustomerID, ContactName, City'

SET @city = '''London'''

SET @sqlCommand = 'SELECT ' + @columnList + ' FROM customers WHERE City = ' + @city

EXEC (@sqlCommand)

**2.Dinamik SQL Sorgusu Çalıştırmak(sp\_executesql())**

Sp\_executesql sistem stored prosedürü Exec() fonsiyonundan farklı olarak, parametrik dinamik sorguların performansını artırmak için tercih edilebilir. Genel kullanımı aşağıdaki gibidir:

Exec sp\_executesql [ @stmt = ] statement

[

{ , [ @params = ] N'@parameter\_name data\_type [ OUT | OUTPUT ][ ,...n ]' }

{ , [ @param1 = ] 'value1' [ ,...n ] }

]

Bir örnekle açıklayalım. Northwind tablosunda adı Nancy olan çalışanın bilgilerine ulaşabilmek için:

declare @sql nvarchar(300), @ado varchar(20), @params nvarchar(30)

set @ado='Nancy'

set @sql='select \* from Employees where FirstName = @ad'

set @params=N'@ad varchar(20)'

execute sp\_executesql @sql, @params, @ad=@ado

**Not:** Girilen değişkenlerin değerinde noktalama işaretleri kullanılacaksa (\*,’, + gibi…) tipleri ‘n’ ile başlamak zorundadır. (ntext / nchar / nvarchar)

Exec() fonksiyonundaki aynı örneği sp\_executesql ile yapalım:

DECLARE @sqlCommand nvarchar(1000)

DECLARE @columnList varchar(75)

DECLARE @city varchar(75)

SET @columnList = 'CustomerID, ContactName, City'

SET @city = 'London'

SET @sqlCommand = 'SELECT ' + @columnList + ' FROM customers WHERE City = @city'

EXECUTE sp\_executesql @sqlCommand, N'@city nvarchar(75)', @city = @city

**SQL ISOLATİON LEVELS**

Ortakzamanlılık(concurrency), aynı kaynağı aynı zamanlarda kullanmak durumunda olma için kullanılan bir terimdir. Örneğin, bir makaleyi ele alacak olursak, teknik editör bu makalenin teknik ayrıntıların geçerliliği üstünde çalışırken, bir başka editör de yazıda bir imla hatası kontrolü yapıyor olabilir. Bu durumda, makaleye ortakzamanlı erişim sözkonusudur. Kaynağa erişecek her istemcinin(editörün) aynı anda aynı işi yapmasa bile ortak bölgelerde bulunması bu durum için yeterlidir.   
  
Kurumsal seviye VTYS’lerle çalışırken, transaction olarak ele alınan işlemlerde en önemli noktalardan biri, ortak zamanlı erişim için geçerli politikaya karar vermektir. Ne zaman ortak çalışmaya izin verilmeli, ne zaman kaynaklar, diğer erişimlere bir süre kapatılmalı gibi sorular bu noktada cevaplanması gereken sorulardır. Bu makalede, ortak zamanlı erişim sorunu ve yaklaşım tarzları ile SQL Server’in sunduğu yöntemler ele alınacaktır.   
  
**Ortak Zamanlı Erişim Sorunları**

Ortak zamanlılığın basit bir tanımından sonra, ortak zamanlı erişimin doğurabileceği neticeleri ele alalım. Ortak zamanlarda aynı kaynağa erişim, yeterli önlemler alınmadığında çeşitli sakıncalar doğurabilir. Bu sakıncalar 4 ana problem olarak standardize edilmiştir:Kayıp Güncelleme, Tekrarlamasız Okuma, Hayalet Okuma ve Kirli Okuma. Bu problemlerden her birini ayrıntılı olarak ele alalım:

* **Kayıp Güncelleme(Lost Update):**Aynı satıra erişen birden fazla transaction olduğunda, bu transaction’lardan verisini en son değiştiren, daha önceki güncellemelerin kaybolmasına neden olur. Örneğin, bir iş anlaşmasını ele elalım. İki yetkili tarafından açılmış ve düzenlemeler yapılıyor olsun. Yetkililerden biri düzenlemeler yapıp kaydettiğinde, ikinci yetkili bu düzenlemeleri asla farketmeyecektir ve ilk kaydetme işleminde birinci yetkilinin yaptığı düzenlemeler kaybolacaktır. Bu sorunu engellemek için, ortak zamanlılığı, birinci yetkilinin işi bitinceye kadar kaynağa erişimi kısıtlayacak şekilde düzenlemek(kilitlemek) gerekir.
* **Kirli Okuma(Dirty Read):**Bir transaction tarafından değiştirilmiş ama henüz kalıcı hale getirilmemiş bir bilginin başka bir transaction tarafından gerçek kayıtmış gibi okunmasıdır. Örneğin, iş anlaşma taslağını ele alacak olursak, yetkililerden biri üstünde değişiklik yapıp henüz işini bitirmediği halde değişikliği kaydettiği anda başka biri tarafından sözleşmenin taraflara iletilmesi, bir kirli okuma problemi doğuracaktır. Bu sorunu engellemek için de yine birinci yetkilinin bütün işlemleri yapıp sözleşmenin son halini kaydetmesine kadar, anlaşmaya erişimi ve dağıtılmasını engellemek gerekir.
* **Tekrarlamasız Okuma(Non-Repeatable Read):**İkinci bir transaction bir satıra her erişiminde farklı bir değer okuyor, sadece okuma yapmasına rağmen bir türlü sabit bir değere ulaşamıyorsa, tekrarlamasız okuma sorunu ortaya çıkar. İş anlaşma metni örneğine dönecek olursak, yetkililerden biri sözleşmenin son halini görmek istediğinde bir göz atıp, kapatıyor. Ardından tekrar bakıyor ve sözleşmenin değiştiğini görüyorsa bu türden bir sorun ortaya çıkacaktır. Sorunu çözmek için, ortak zamanlılığı şu şekilde düzenlemek gerekir: Sözleşme taslak halde iken, yani üstünde biri düzeltme yaparken, sözleşmenin son halini görmek isteyen erişimleri engellemek(kilitleme) gerekir.
* **Hayalet Okuma(Phantom Read):**Buraya kadar incelediğimiz sorunlarda hep transaction’lardan biri okuma yaparken, diğeri güncelleme yapıyordu. Oysa veritabanında her zaman güncelleme yapılmaz. Bazen silme ve ekleme de yapılır. İşte Transaction’lardan biri silme veya ekleme(Insert-Update) işlemeri gerçekleşirken diğer bir transaction bu işlemleri içine alan aralıktaki satırlar üstünde okuma yapıyorsa, ya elde ettiği satırlardan bir kısmı artık yoktur veya elde ettiği satırlar, bazı satırları eksik bulunuduruyordur. İş anlaşmasındaki hali ile, sözleşme taslağı üstünde taraflardan biri değişiklikler yapıp avukatına gönderdiğinde, avukatı bu değişikliklerin asıl sözleşme ile uyuşmadığını belirtir. Durum ayrıntılı olarak incelendiğinde, diğer taraftan yeni bir sözleşme metni maddesi eklendiği veya bir maddenin çıkartıldığı için böyle bir sorunun oluştuğu anlaşılırsa hayalet okuma sorunu ortaya çıkmış olur. Bu türden durumlara karşı ortak zamanlı çalışmayı düzenlemek için, orinal sözleşme ikinci tarafa ulaştıktan sonra, asıl nüshası üstünde ikinci tarafın işi bitinceye kadar, birinci tarafın asıl kopya üstünde düzenleme yapmamasını sağlamak(kilitleme) gerekir.

Görüldüğü gibi, ortak zamanlı erişimde meydana gelecek problemlerin çözümünde en temel nokta, erişimi bir süre için bütün başkalarının veya sadece okuyacakların veya sadece değiştireceklerin erişimine karşı engellemek gerekmektedir. Bu işleme kilitleme(locking) denir.

**Ortakzamanı Erişim İzolasyon Seviyeleri**

SQL-92 standardı, ortakzamanlı erişim esnasında kullanılabilecek 4 izolasyon seviyesi öngörür: 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **İzolasyon Seviyesi** | **Açıklama** | **Kirli Okuma** | **Tekarlamasız Okuma** | **Hayalet Okuma** |
| **Read Uncommitted** | Bir transaction okuma yaparken, diğer transaction’lara karşı hiç bir kilitleme yapmadığından, diğer transaction’lar verileri değiştirebilir. En basit izolasyon seviyesidir. Veri tutarlılığı uygulamaya bırakılmıştır. | + | + | + |
| **Read Committed** | SQL Server’in default yaklaşımıdır. Bir okuma işlemi için erişen transaction işini bitirmeden, veriler değiştirilebilir. | - | + | + |
| **Repeatable Read** | Kirli okuma ve Tekrarlamasız okumalara karşı kilitleme mekanizmasının olduğu bir izolasyon seviyesidir. | - | - | + |
| **Serialisible** | Veri eklerken, silinirken veya güncellenirken transaction boyunca veriler erişime kapatılır. Hiç bir ortak zamanlılık sorunu meydana gelemez ancak işlemler çok bekler ve deadlock’lar meydana gelebilir. | - | - | - |

SQL Server ortakzamanlılığı düzenlemek için bu standartca tanımlanan izolasyon seviyelerine destek sağlar.

**Transaction Nasıl Çalışır**

SQL Server, veri üstünde değişiklik yapacağı zaman, Şayet daha önceden tampon belleğe aktarılmış bir sayfa değil ise, öncelikle bu verilerin üstünde yer aldığı sayfa(page)ları disk(HDD)’ten hafızaya(RAM)’da bir bölgeye taşır. Bu bölgeye tampon hafıza(buffer cashe) adı verilir. Ardından veriler üstünde ilgili değişiklikler yapılır. Üstünde değişiklik yapılmış ve henüz kalıcı olarak diske kaydedilmemiş sayfalara kirli sayfa(dirty page) denir.   
  
Kirli bir hafıza ile ilgili değişim, transaction dosyaları (\*.ldf uzantılı dosya)na işaretlenir. Sayfaların diske aktarılması esnasında doğacak bir soruna karşı önlem olarak değişiklikler hep bu şekilde öncelikle loglara yansıtılır. Ardından kirli sayfalar diske aktarılır. Bu işleme de arıtma(flushing) denir.

**Transaction İçin İzolasyonu Belirlemek**

Bir transaction için, SQL Server’in yukarıda belirtilen izolasyon seviyelerinden birinde işlem yapmasını isteyelim:   
  
Bunun için genel ifade şu şekildedir:   
  
SET TRANSACTION ISOLATION LEVEL izolasyon\_seviyesi   
  
Bu ifadede, *izolasyon\_seviyesi*parametresinin yerine, yukarıdaki tabloda verilen parametrelerden biri gelebilir.   
  
Eş zamanlı olarak çalışan Transaction' larda meydana gelebilecek bu problemleri, IsolationLevel numaralandırıcısı  yardımıyla kontrol altına alabiliriz. Bu numaralandırıcının alabileceği değerler ve bu değerlerin izin verdiği(vermediği) durumlar aşağıdaki tabloda yer almaktadır.

|  |  |  |  |
| --- | --- | --- | --- |
| **IsolationLevel Numaralandırıcı Değeri** | **Olası Problemler** | | |
| **Phantoms** | **Non-Repeatable Read** | **Dirty-Read** |
| **Chaos** | SqlServer tarafından desteklenmez. | | |
| **ReadCommitted** | http://www.buraksenyurt.com/makale/images/mk75_1.gif | http://www.buraksenyurt.com/makale/images/mk75_1.gif | http://www.buraksenyurt.com/makale/images/mk75_2.gif |
| **ReadUncommitted** | http://www.buraksenyurt.com/makale/images/mk75_1.gif | http://www.buraksenyurt.com/makale/images/mk75_1.gif | http://www.buraksenyurt.com/makale/images/mk75_1.gif |
| **RepeatableRead** | http://www.buraksenyurt.com/makale/images/mk75_1.gif | http://www.buraksenyurt.com/makale/images/mk75_2.gif | http://www.buraksenyurt.com/makale/images/mk75_2.gif |
| **Serializable** | http://www.buraksenyurt.com/makale/images/mk75_2.gif | http://www.buraksenyurt.com/makale/images/mk75_2.gif | http://www.buraksenyurt.com/makale/images/mk75_2.gif |
| **Unspecified** | SqlServer tarafından desteklenmez. | | |

Görüldüğü gibi IsolationLevel numaralandırıcısının alabileceği değerler altı adettir. Chaos ve Unspecified numaralandırıcı değerleri sql server tarafından desteklenmemektedir. Bununla birlikte, bir Transaction sınıfının IsolationLevel özelliğinin varsayılan değeri ReadCommitted olarak belirtilmiştir. Bu tablonun okunuşuna gelince. Örneğin, RepeatableRead değerini ele alalım. Bir uygulamada, Transaction nesnesine izolasyon seviyesi olarak bu değer atandığı takdirde, eş zamanlı olarak çalışan Transaction' lar arasında sadece Phantoms durumuna izin verilir. Dolayısıyla, Non-repeatable Read ve Dirty-Read durumlarına izin verilmez. Burada tüm olası problemlere izin veren IsolationLevel numaralandırıcı değeri ReadUncommitted değeridir. Aksine, Serializable değeri Transaction' lar arasında doğabilecek bu problemlerin hiç birisinin olmasına izin vermez.

Bunlara ek olarak Mssql in bizlere sağladıkları ise ;  
1-) Read Commited Snapshot Isolation  
2-) Snapshot Transaction Isolation

## Read Commited Snapshot Isolation

      Bu level da Read Commited level da oluşan **Shared Lo**ck durumu ortadan kalkar. Yani bir transaction bir veride güncelleme yaparken bir başka transaction bu veriyi okuyabilir. Bir transaction içerisinde kalmış ve commit edilmemiş verilerin **tempdb** veritabanına snapshotlarını kaydeder. Böylece diğer transactionların verileri okuması sağlanmaktadır.

Kullanılabilmesi için database seviyesinde **READ\_COMMITTED\_SNAPSHOT** parametresinin "on" konumunda olması gerekmektedir. Default olarak "off" ayarlıdır.

ALTER DATABASE Northwind SET READ\_COMMITTED\_SNAPSHOT ON

olarak kullanılır.

## Snapshot Transaction Isolation

    Bir üstteki level dan tek farkı burada yapılan konfigürasyonun transaction bazında olmasıdır. Kullanımında öncelikle database seviyesinde ayar yapılması gerekir.

ALTER DATABASE Northwind SET ALLOW\_SNAPSHOT\_ISOLATION ON

Daha sonra ise transaction seviyesinde ayarlama yapılması gerekmektedir.

SET TRANSACTION ISOLATION LEVEL SNAPSHOT

**İzolasyon Sağlama**

Farklı transaction’ları aynı anda yönetirken, erişilen kaynaklardaki değişimlerin bir diğer transaction tarafından görünebilirliğini veya aynı anda yansıtılabilir olmasını ayarlamak için iki temel yöntem kullanılır:

* **Kilitleme (Locking)**

Kilitleme temelli olarak sağlanan ortak zamanlılıkta, her bir transaction farklı büyüklükte kaynak kilitleyebilir. Bir transaction bitimine kadar, bir satırı, bir sayfayı veya bir tablonun tamamını başka transaction’ların erişimine kapatılabilir.

* **Satır Versiyonlama**

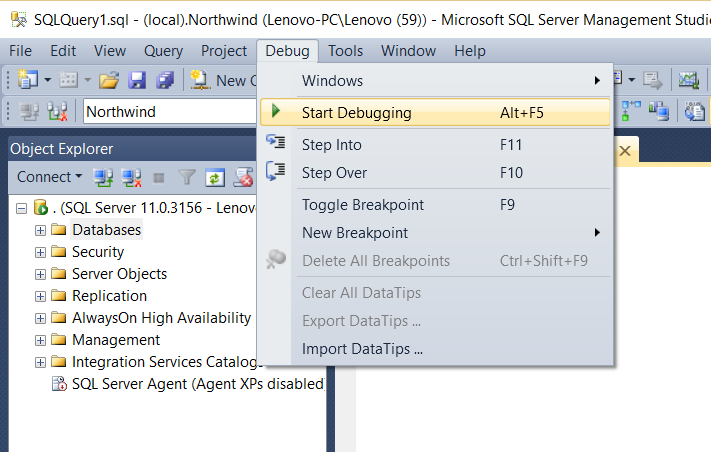
Satır versiyonlama ise satırların eski versiyonlarının tempdb veritabanında saklanmasıdır. Satır güncellenmeden önce eski versiyon transaction tarafından okunabilir. Böylece bazı verileri kilitlemeye gerek duyulmaz.

**SQL DEBUGGING**

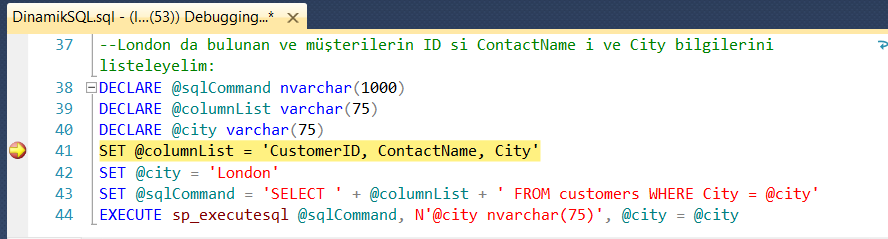
Sql kodlarının hata verdiği durumlarda ya da beklenmedik değerler aldığımızda hatta yeni başlayanların kodların nasıl çalıştığını daha iyi anlayabilmek adına debug işlemleri programcı için büyük kolaylıklar getirmektedir.

**Debug** menüsünden **Start Debugging** ‘e tıklayarak veya **ALT+F5** yaparak Debug ekranımızı açıyoruz.

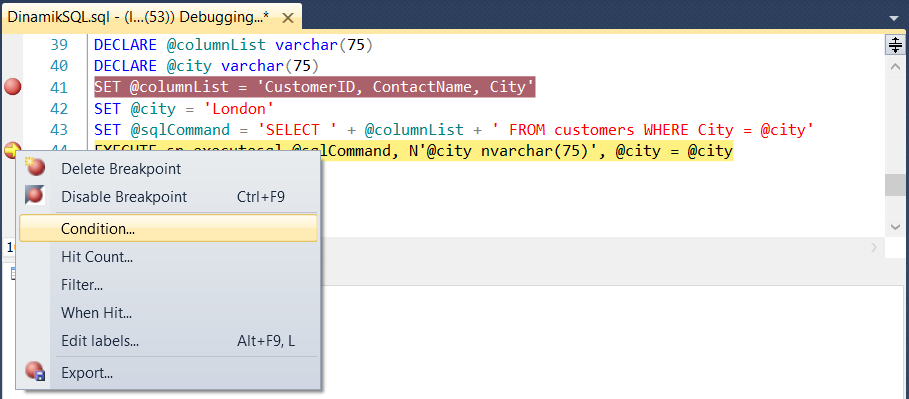
|  |  |
| --- | --- |
| F11 | Eğer o esnada üzerinde bulunan kod satırı, procedure veye metod değilse o satırı çalıştırıp, bir alt satıra geçip çalışma tekrar duracaktır. Eğer metod ise metodun içine girip metodun ilk satırında çalışma duracaktır. |
| F10 | Step Into ile benzer, tek farkı eğer kod satırı bir metod çağırıyorsa metodun içine girmez, metodun hepsini çalıştırıp diğer satıra geçer. |
| Continue - F5 | Debug etmeyi durdurur, Normal çalışmaya devam eder. |



Burada SELECT ifademizin üzerine gelip **F9**‘a basıp breakpoint koyuyoruz.



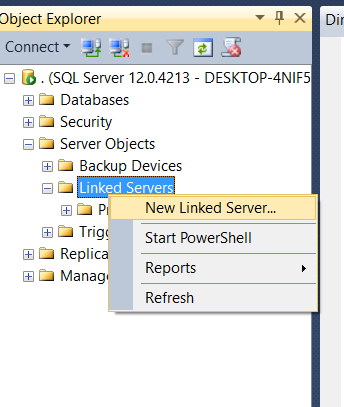
 Breakpoint’imizin üzerine sağ tıklayıp **Condition** ‘a giriyoruz.

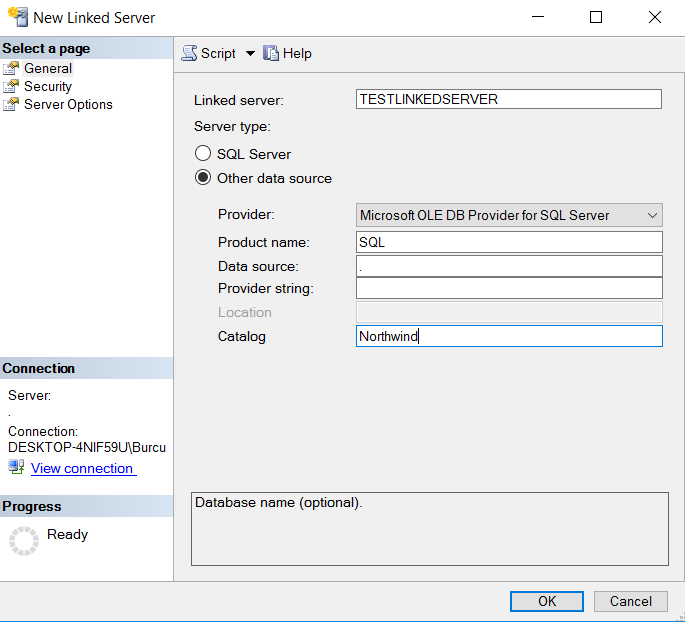


**SQL LINKED SERVERS(BAĞLI SUNUCULAR)**

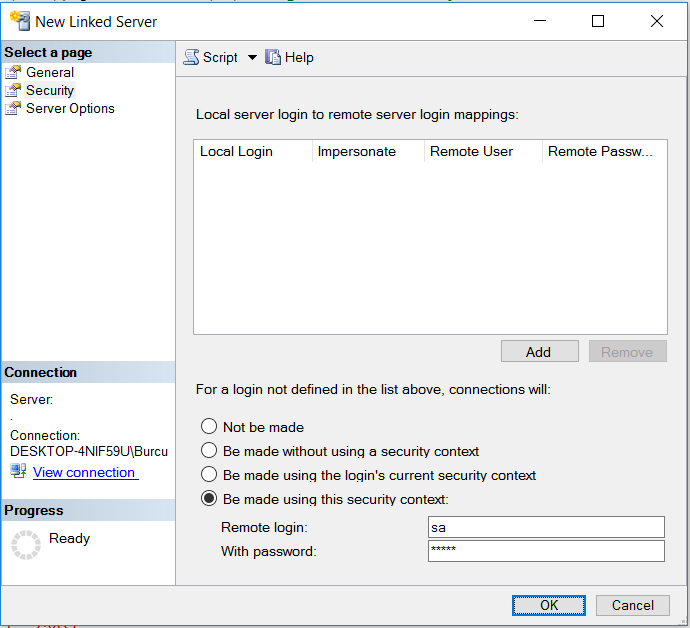
SQL Server üzerinde farklı veritabanlarını kullanarak işlem yapmak mümkün.Aynı Server üzerinde bu işlemi kolayca yaparken network üzerindeki farklı serverlara erişebilmek için Linked Server kullanılır.

Öncelikle SQL SERVER a girip Server a bağlanıyoruz. Aşağıdaki adımları sırayla gerçekleştirelim.

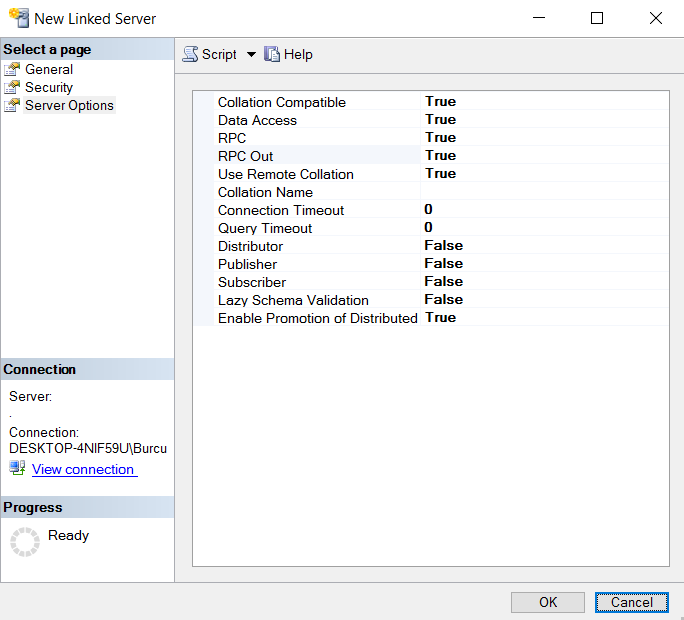


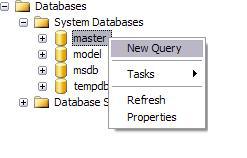


Linked Server kısmına Linked server için herhangi bir isim veriyoruz. Provider kısmına SQL SERVER ile bağlantı kuracağımız için Microsoft OLE DB Provider for SQL Server seçiyoruz. Product Name olarak “SQL” yazdıktan sonra Datasource kısmına eğer benim gibi aynı makine üzerinden işlem yapacaksanız sol tarafta Connection başlığı altında yer alan Server ismini, Network üzerinden başka bir PC deki SQL SERVER a bağlanacaksanız ya o server ın ismini ya da IP sini giriyorsunuz. Son olarak da Catalog kısmına diğer server da kullanacağınız veritabanının ismini yazıyoruz.



Eğer aynı server da işlem yapıyorsanız “Be made using the login’s current security context” seçili kalabilir. Fakat başka bir Server a bağlanıyorsanız ki Linked server ın amacı budur, “Be made using this security context” seçilip bağlandığınız SQL SERVER ın kullanıcı adı ve şifresini yazıyoruz.



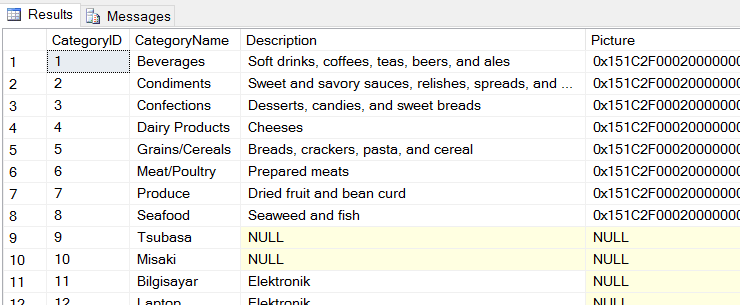
Yukarıdaki alanları true yapmanız yeterli olacaktır.

Son olarak ben işlem yaptığım veritabanını System Databases içerisinde bulunan master database i seçtim ve üzerine gelip New Query seçiyorum.

Northwind Veritabanı üzerinde önceden kullandığım “Categories” tablosunu kullanıyorum.  
Eğerki Northwind veritabanı üzerinden işlem yapıyor olsaydım.

select \* from Categories

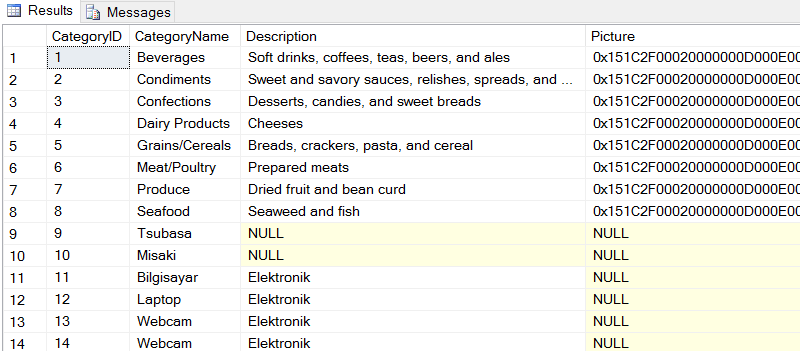
şeklinde kullanıyor olacaktım ve sonucunu şu şekilde alırız;



Fakat Master veritabanı üzerindeki bir query den Northwind veritabanındaki bu tabloya ulaşmak istersem Linked Server kullanarak şu şekilde sorgumu oluşturuyorum

Select \* From Openquery(TESTLINKEDSERVER,'select \* From Categories')

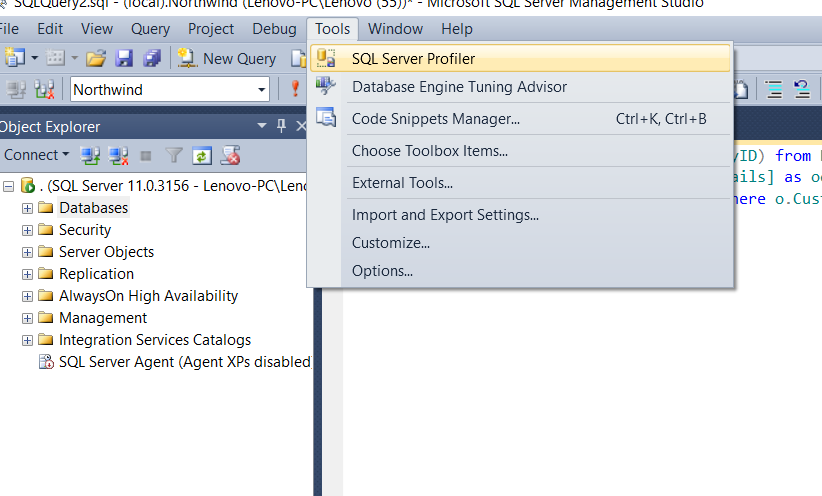
ve bu sorgumun sonucu da diğerinden farksız bir şekilde aynı sonucu verecektir.



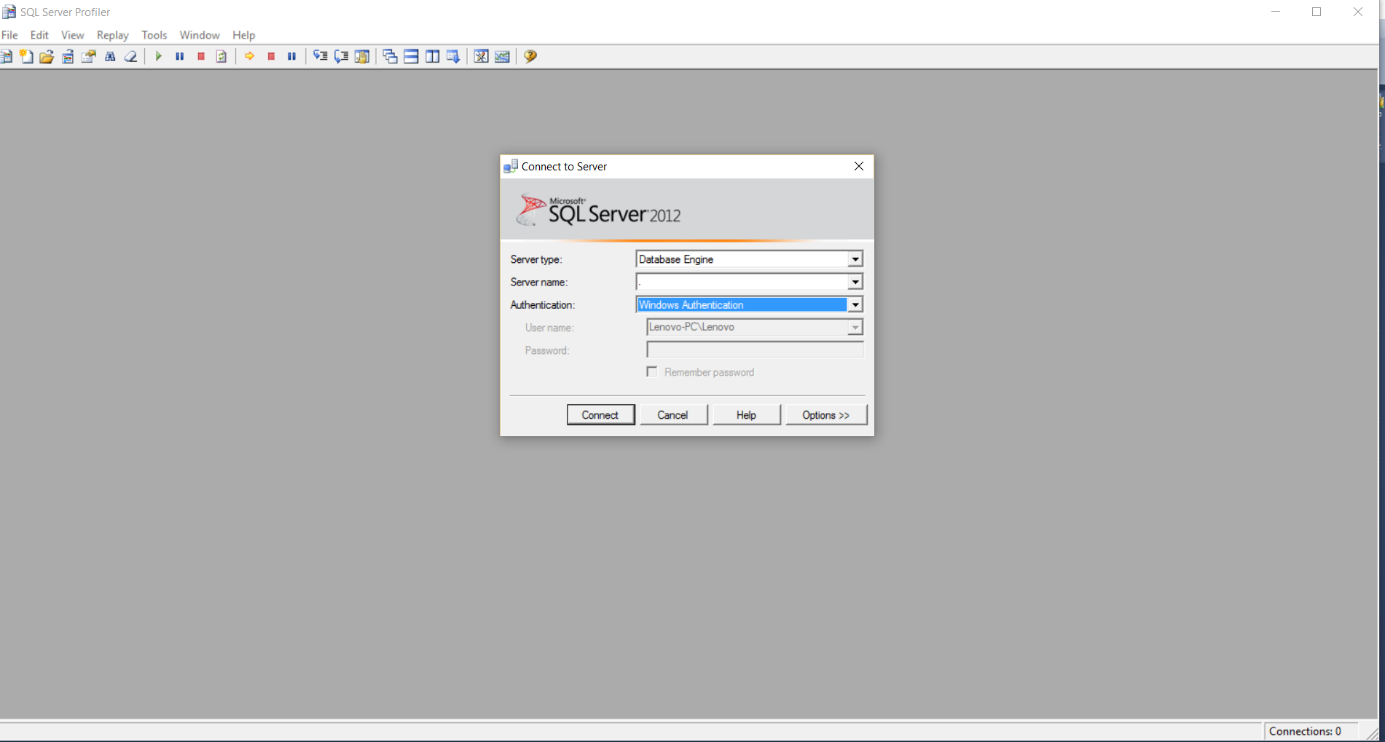
**SQL SERVER PROFILER**

Sql Server içerisinde bu durumu izleyebildiğimiz ve süreci yönetebildiğimiz kısım Sql Server Profiler’dır. Bu araçla SQL Server üzerinde gerçekleşen onlarca olayın performansını izler ve potansiyel yaşanan sıkıntıları görüntüler ve bize bilgi verecek bir veritabanı içerisinde tablo veya trace file 'ı oluşturmaya yarar.

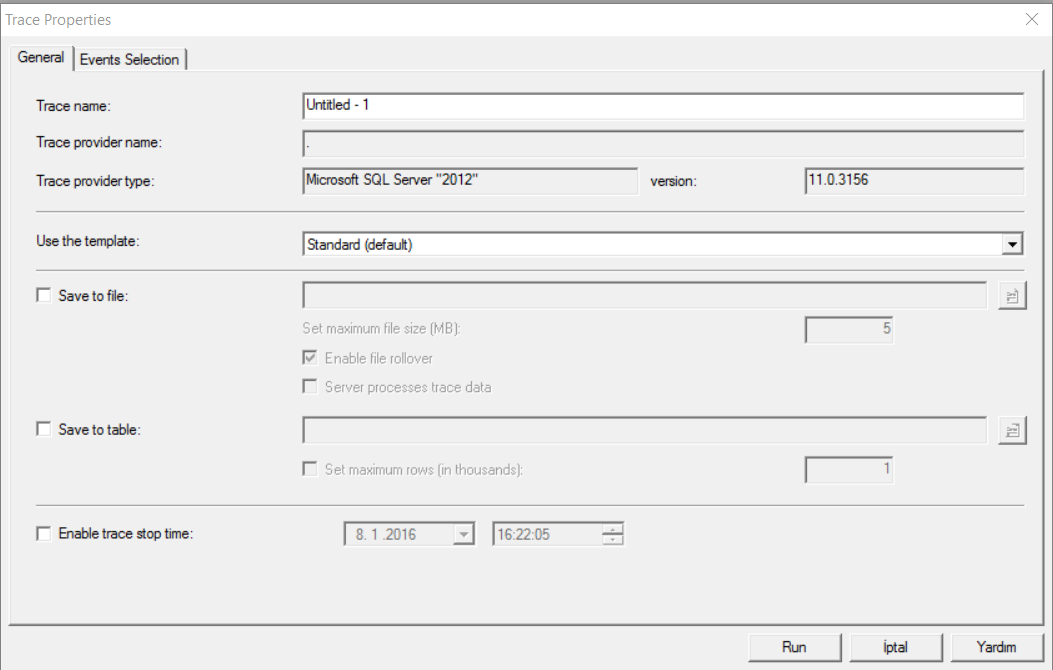
Profiler’ ı kullanabilmek için:



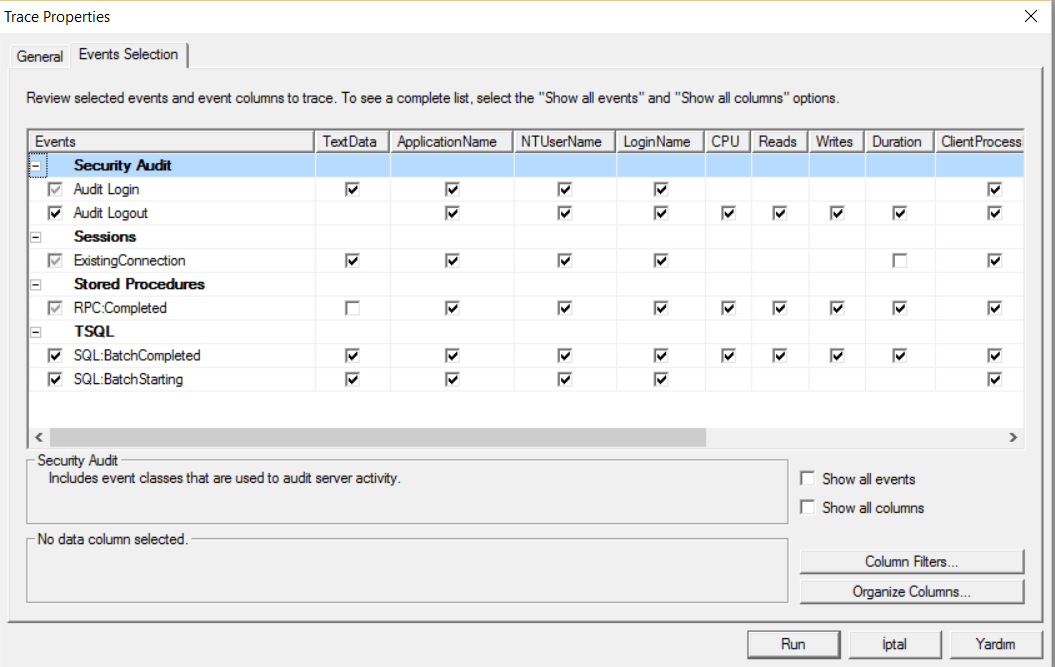
SQL Server a login girişi yapılır.



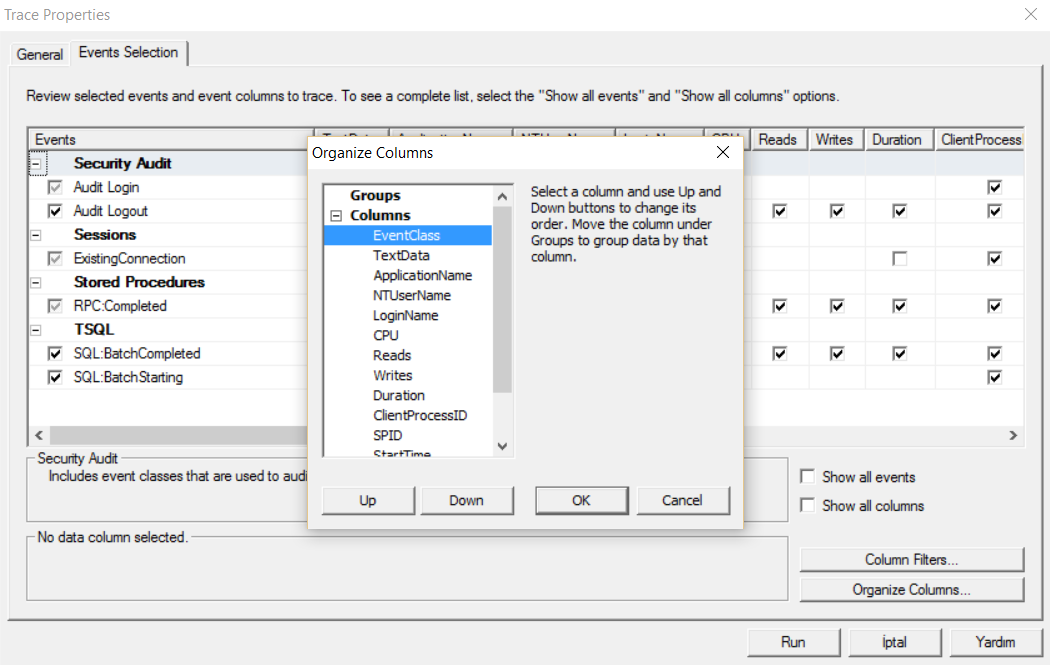
Giriş yaptıktan sonra;

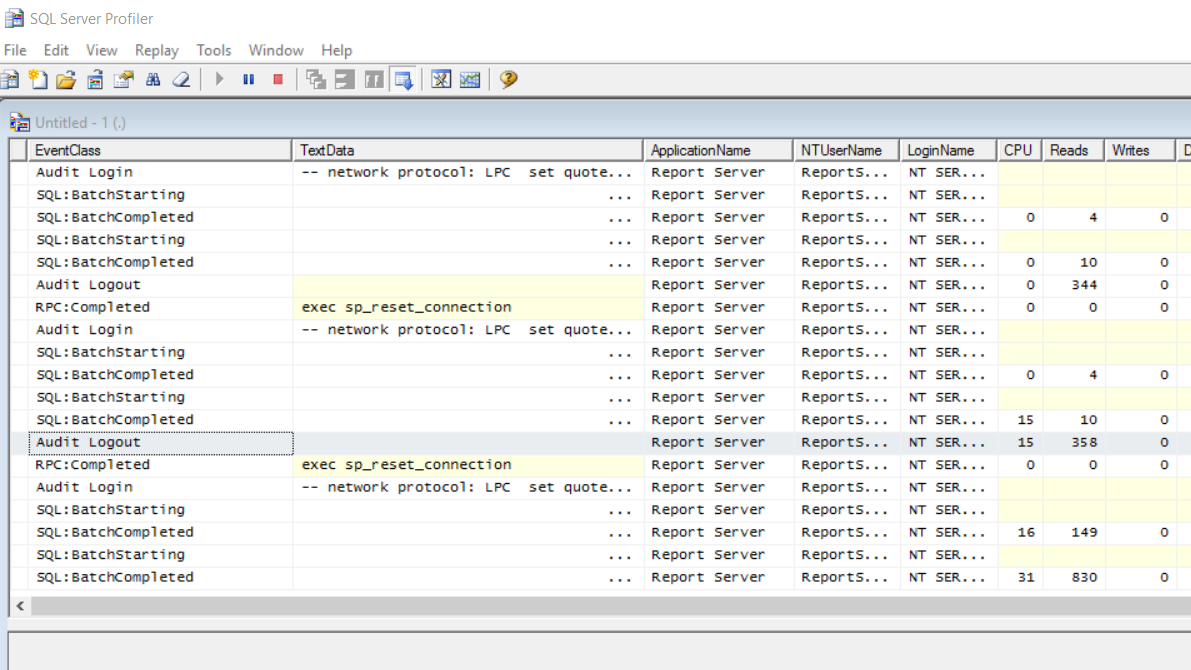


Bu islemin ardindan Sql Server'imizda hangi olaylari yada hangi islemleri izleyecegimizi belirttigimiz Event Selection tabinda istenilen ve ihtiyac duydugumuz olaylari seciyoruz.Burada onemli olan nokta alt kisimda bulunan Show All Events isimli secenegi aktif hale getirerek sunucumuz uzerinde bize sunulan tum secenekleri gorebiliriz.



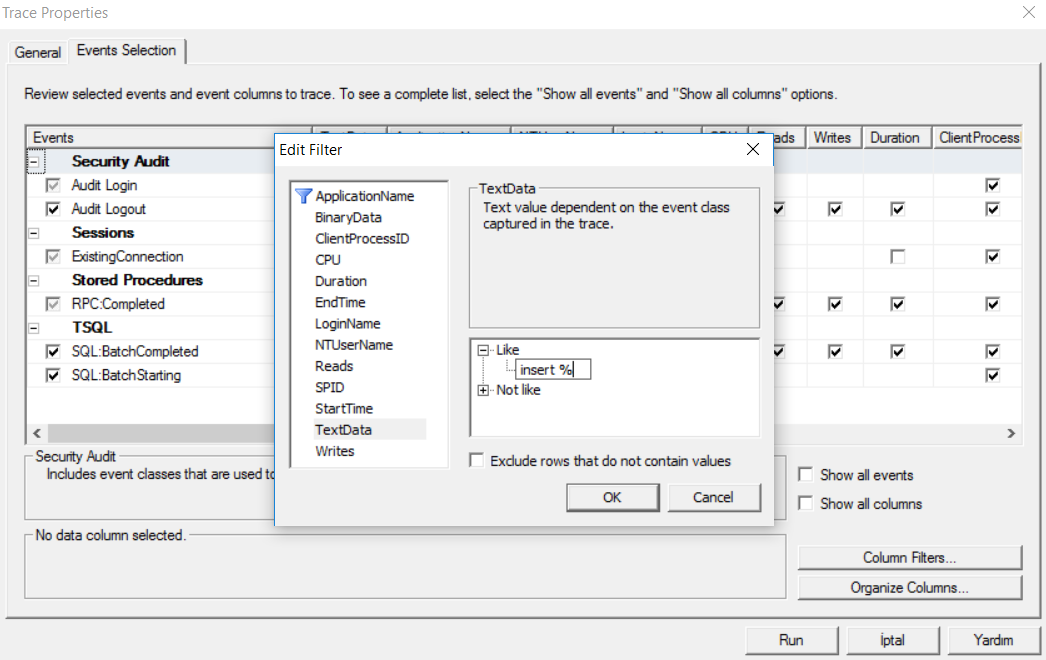
Bu islemin ardindan Organize Columns secerek trace ettigimiz bilgierin neler olacagi ve hangi isleme gore gruplanacagini secebiliriz.



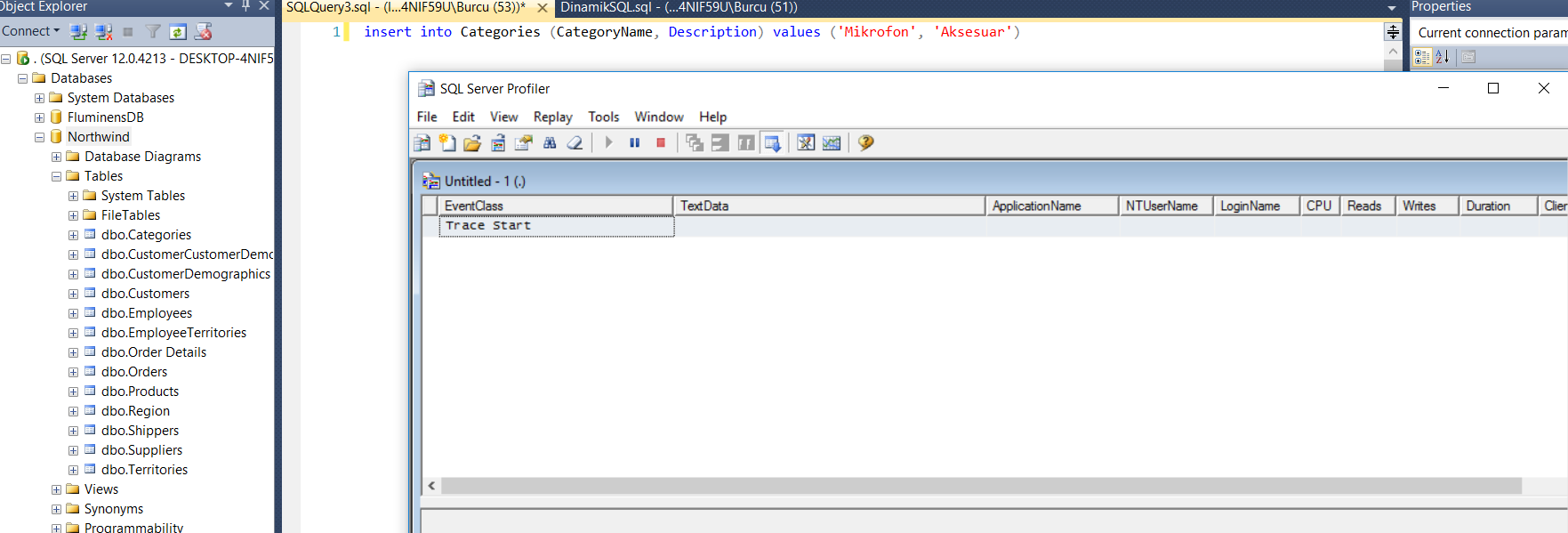
Bu gruplamayida yaptiktan sonra istersek Column Filter ozelligi ki cok fazla onemli hangi db uzerinde trace islemi uygulayacagimizi aksi halde bu sunucuda ki gerceklesen tum islemler icin trace islemi uygulayacagini bilmemiz gerekir.Gerekli sarti da bu sekilde verebiliriz.  
Ardindan Run dersek artik sunucumuz izlenmeye baslanacaktir.

Profiler’ ı gerçek anlamda projelerin test aşamasında kullanabiliriz.

Şimdi bir insert işlemini izleyelim. Yine aynı şekilde Tools menüsünde SQL Server Profiler ı seçip açılan pencereden server a bağlantı kuruyoruz. Event Selection tabindan Column Filter butonuna tıklıyoruz. Açılan pencerede **TextData** kısmına gelip **Like** altına izleyeceğimiz **insert** keyword ünü yazıyoruz. **%** işareti, koymamızın sebebi, sonu ne olursa olsun insert ile başlayan sorguları bul, deyip **OK** diyorum. Ve Trace i çalıştırıyorum.



Şimdi yazdığımız bir insert sorgusunu çalıştıralım.



Sorguyu çalıştırdığım da Profiler daki görüntü aşağıdaki gibidir.

