AĞ TABANLI PARALEL DAĞITIM SİSTEMLERİ VİZE PROJELERİ

AHMET EREN ÖZCAN

21290207

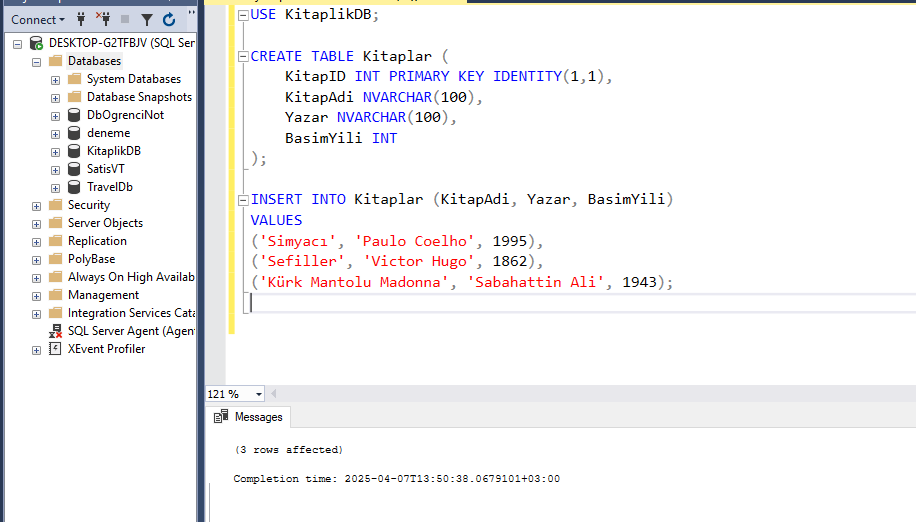
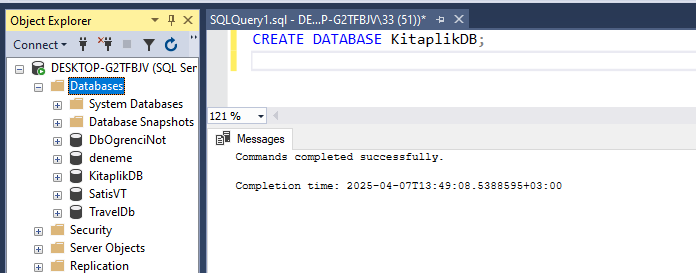
GİTHUB:ahmeterenozcan

**1-) SQL Server Veritabanı Yedekleme ve Otomasyon Projesi**

**Proje Adı:** KitaplikDB Günlük Yedekleme

**Adım 1: Veritabanı Oluşturma**

**Yapılan İşlem:**  
SSMS üzerinde KitaplikDB adlı bir veritabanı oluşturuldu. Daha sonra örnek bir Kitaplar tablosu eklendi ve örnek veriler girildi.

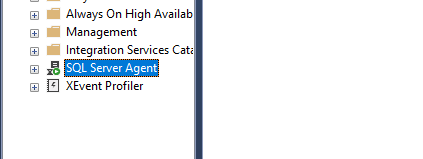


**Adım 2: Yedekleme Klasörü Oluşturma**

**Yapılan İşlem:**  
D:\SQL\_Backups\ klasörü oluşturuldu. SQL Server servis hesabının bu klasöre erişim izni verildi.

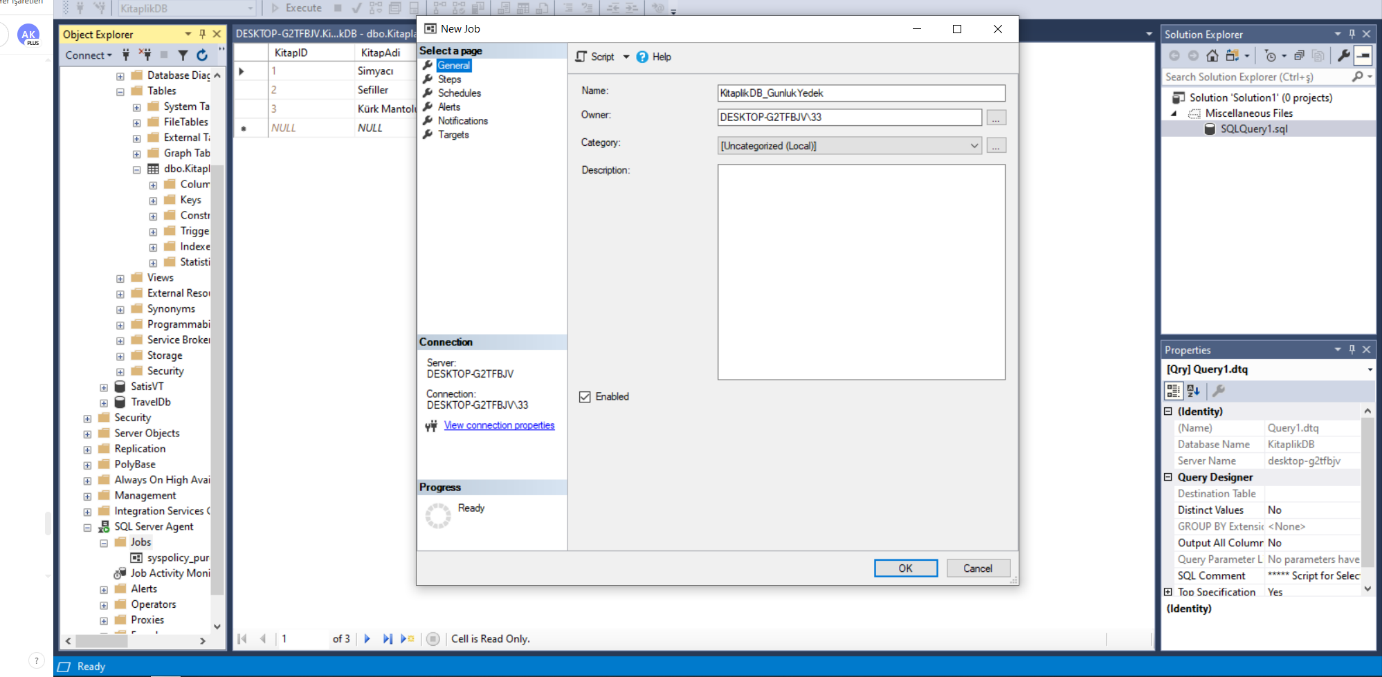
**Adım 3: SQL Server Agent’ı Başlatma**

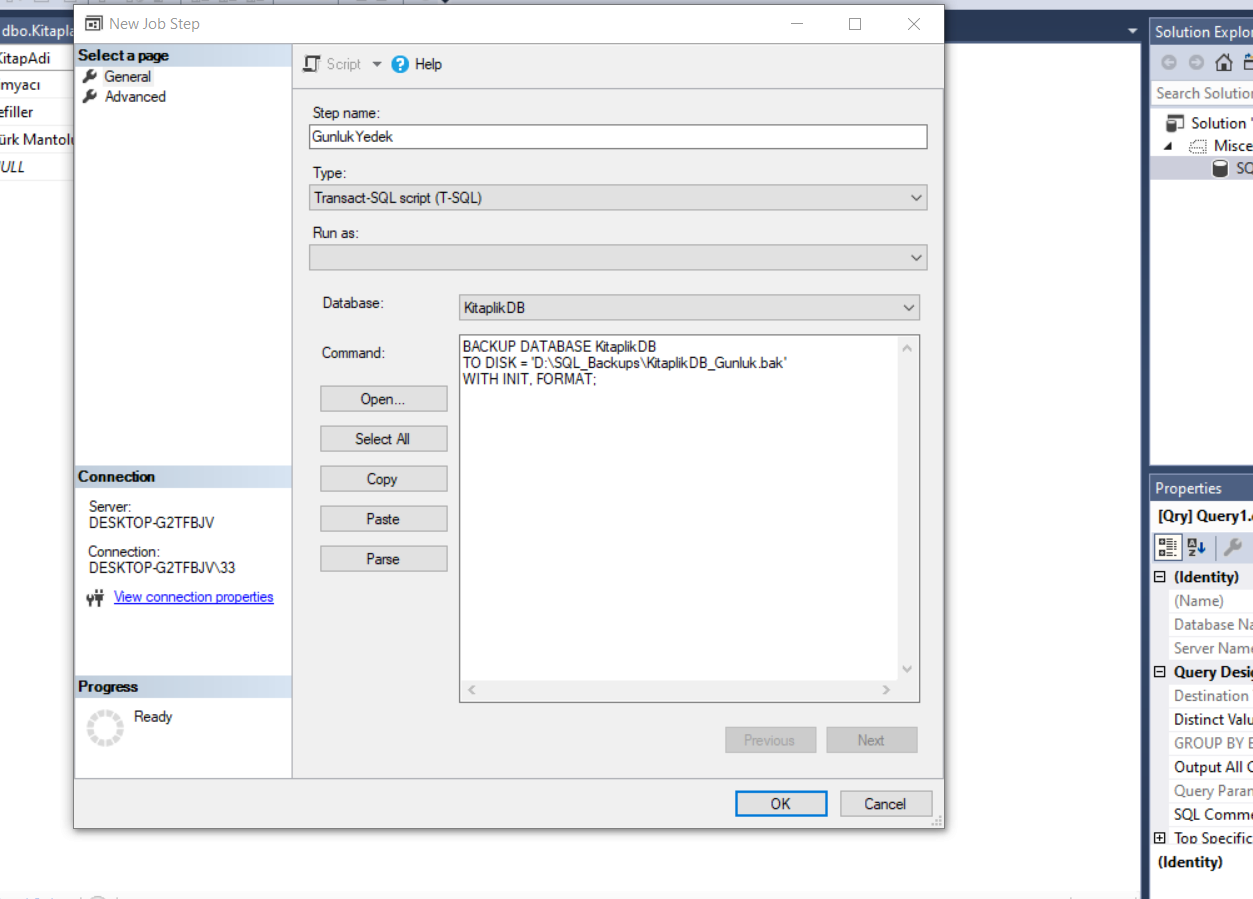
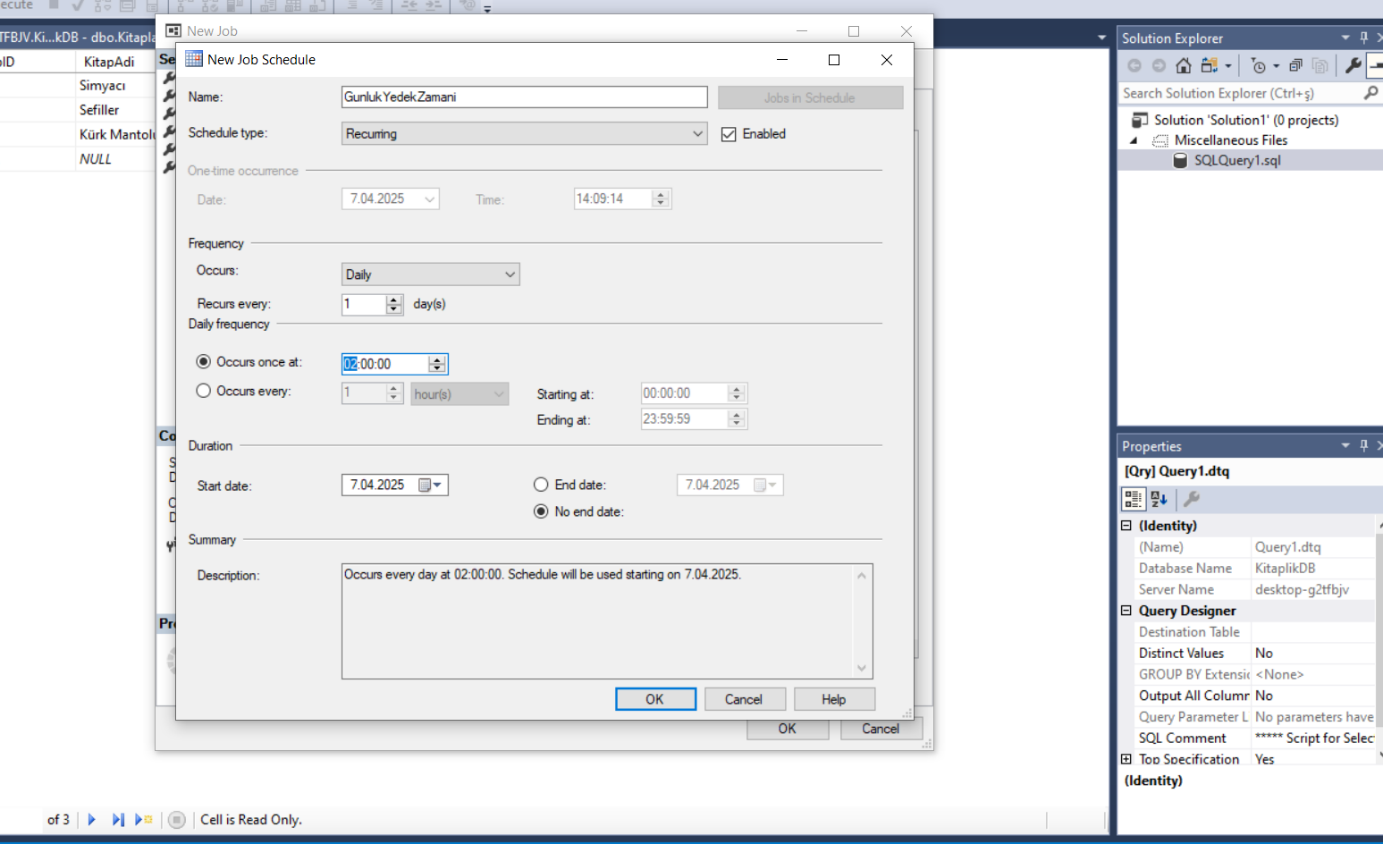
**Yapılan İşlem:**  
SQL Server Agent servisi başlatıldı. Bu sayede zamanlanmış görevler (Job'lar) çalıştırılabilir hale geldi.



**Adım 4: SQL Server Agent Job Oluşturma**

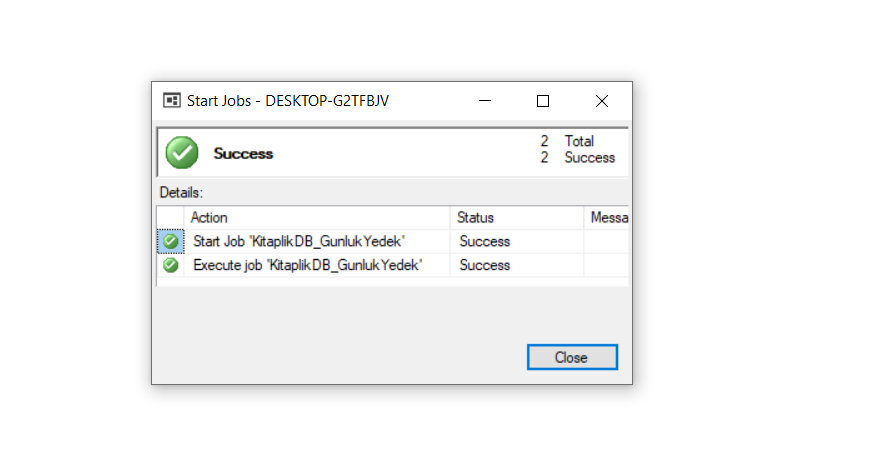
**Yapılan İşlem:**  
Bir Job oluşturularak, KitaplikDB veritabanı her gün otomatik olarak yedeklenecek şekilde ayarlandı.

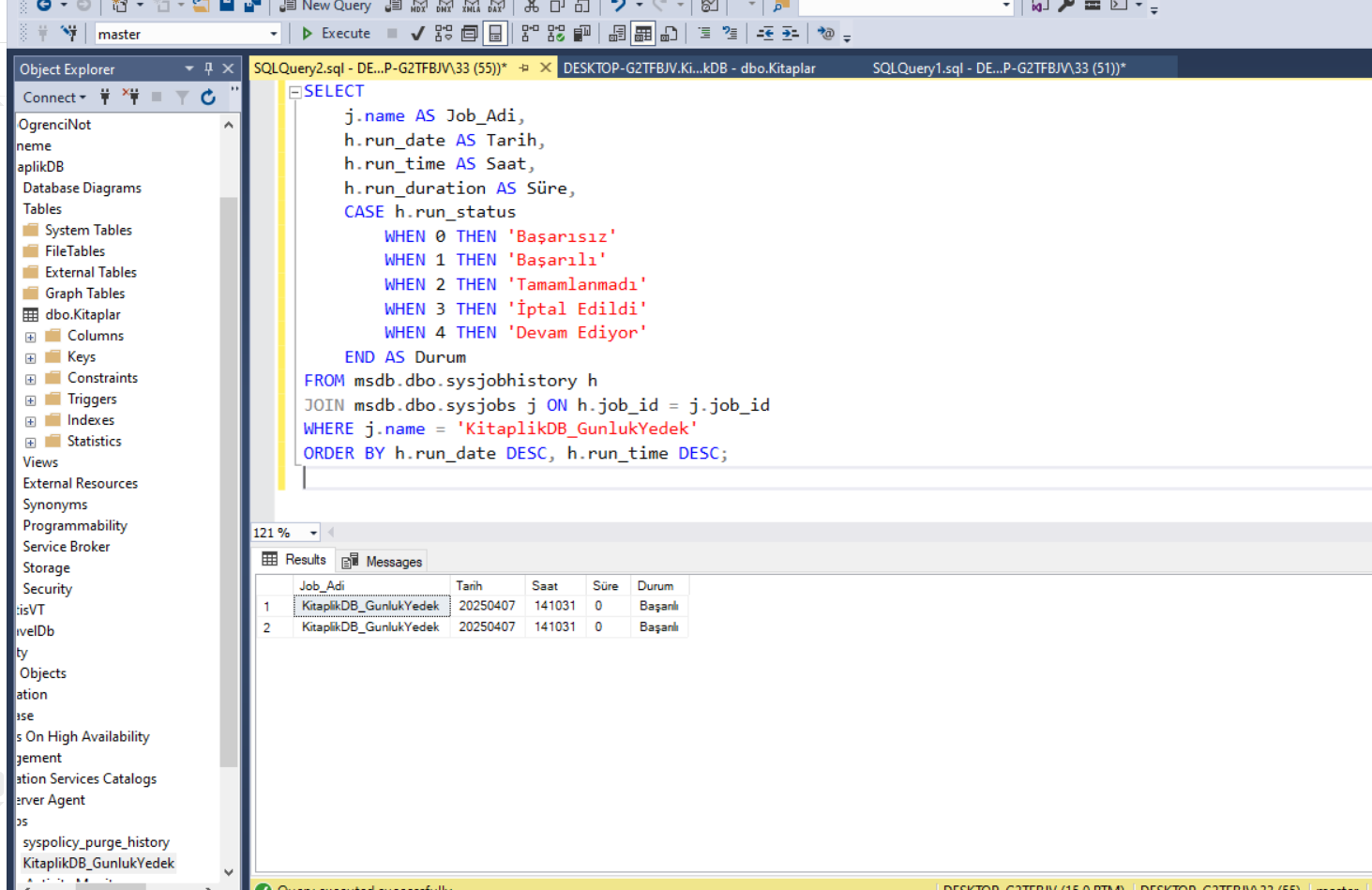




**Adım 6: Yedekleme Raporu Oluşturma**

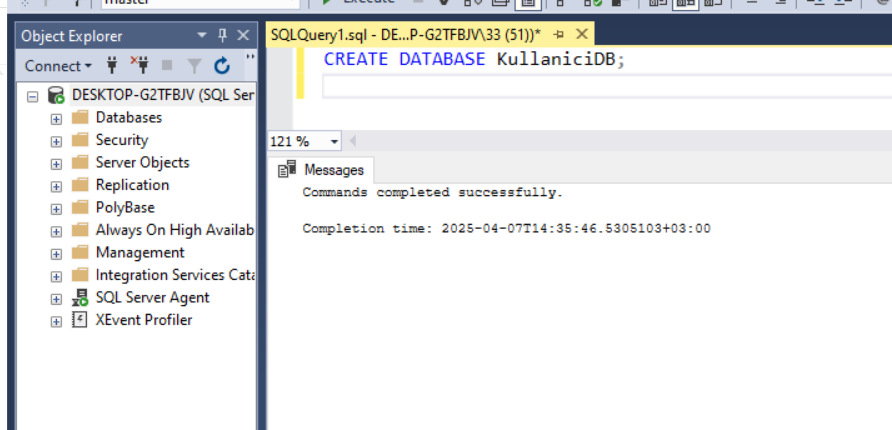
**Yapılan İşlem:**  
Job geçmişi sorgulandı ve yedeklemenin başarılı olup olmadığı izlendi.

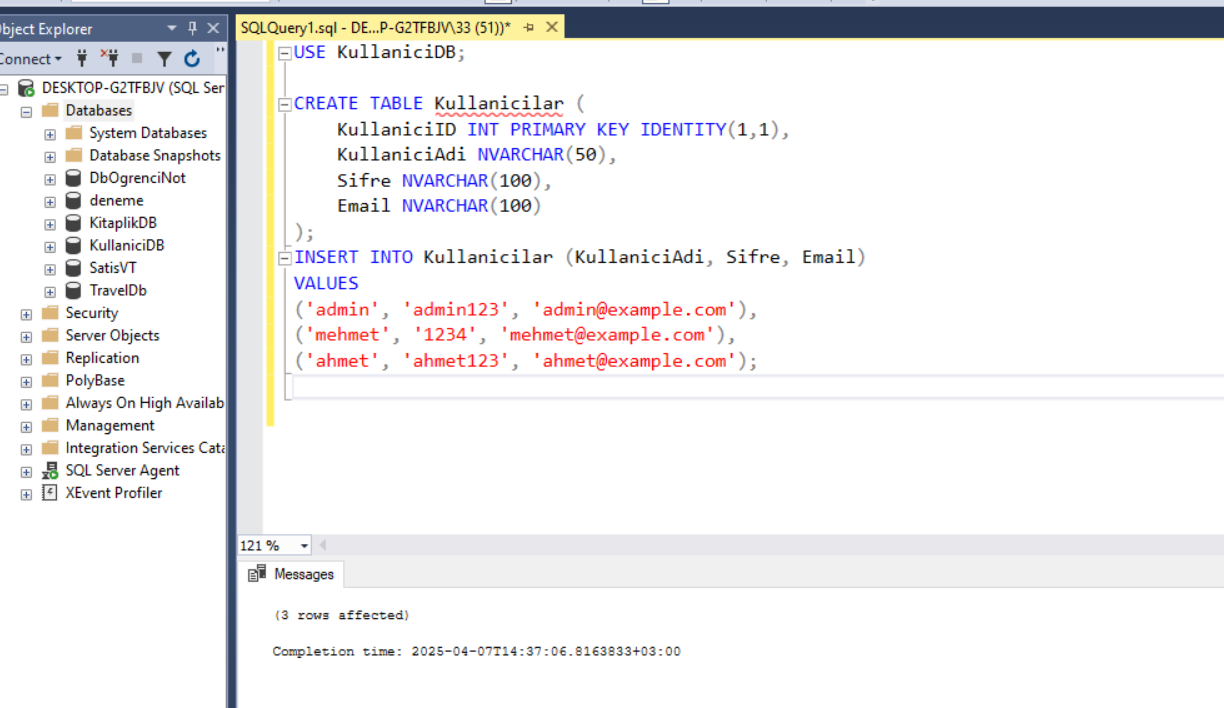




2-)Veritabanı Güvenliği ve Erişim Kontrolü

**Adım 1 – Veritabanı ve Tablo Oluşturma**

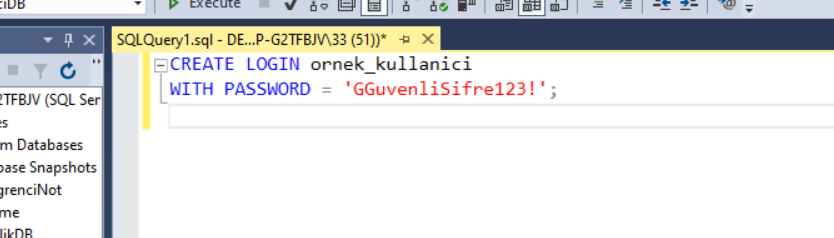
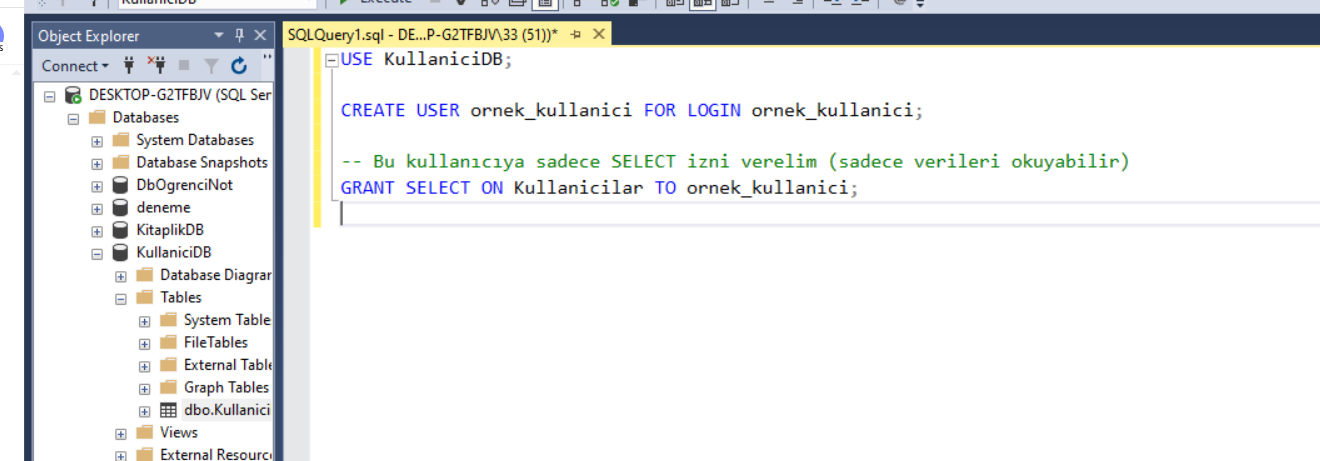
****KullaniciDB isimli bir veritabanı ve içerisinde Kullanicilar adlı tablo oluşturulmuştur.  
Tabloya örnek kullanıcı verileri girilmiştir.

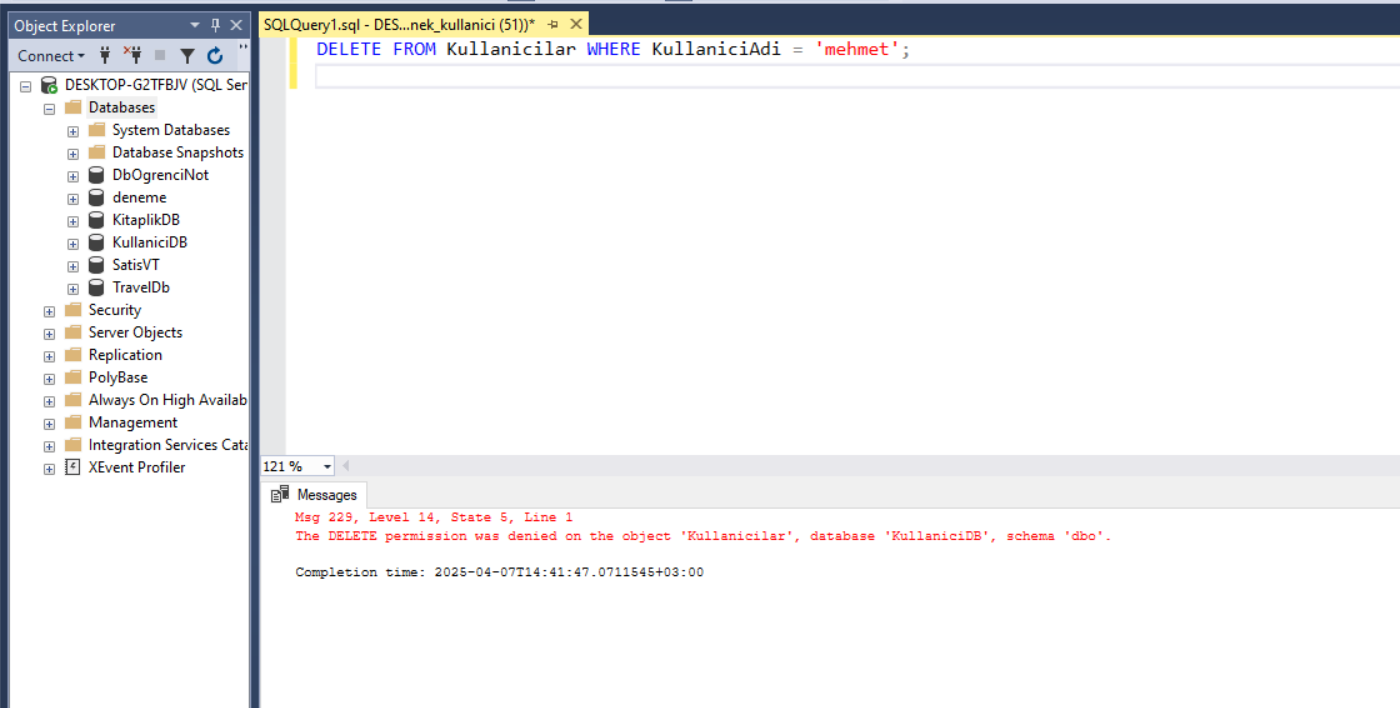


**Adım 2 – Kullanıcı Oluşturma ve Yetki Verme**

**Açıklama:**

ornek\_kullanici adlı bir SQL Server Login oluşturuldu ve sadece SELECT izni verildi.



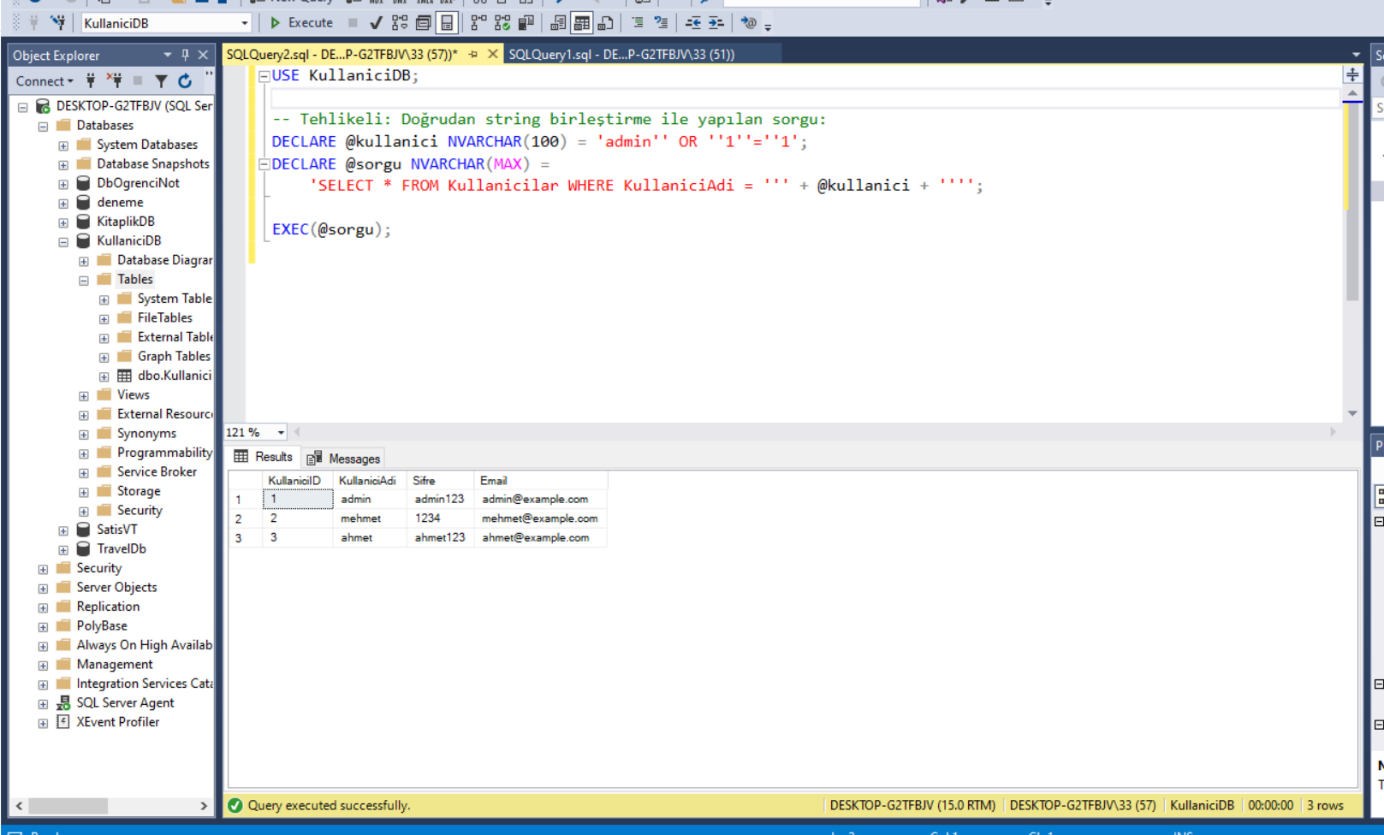
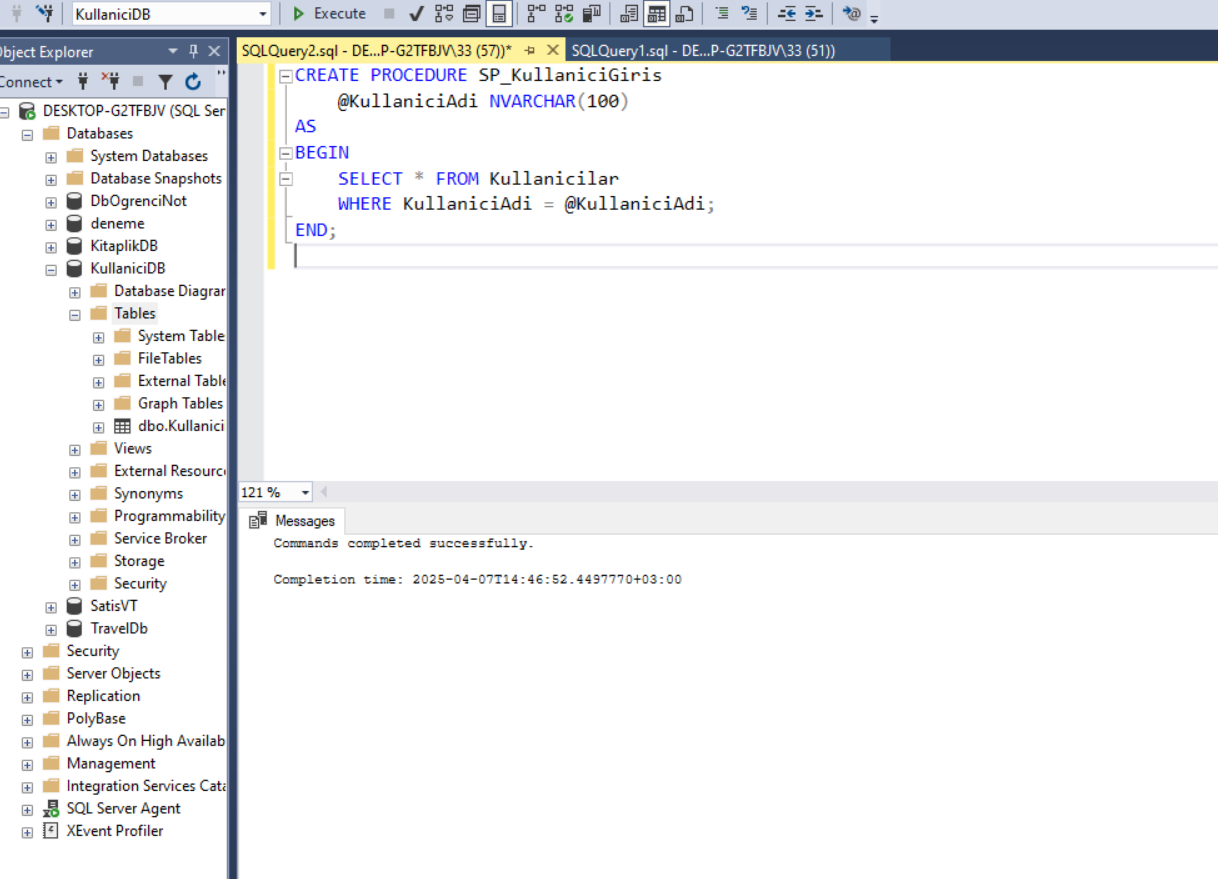


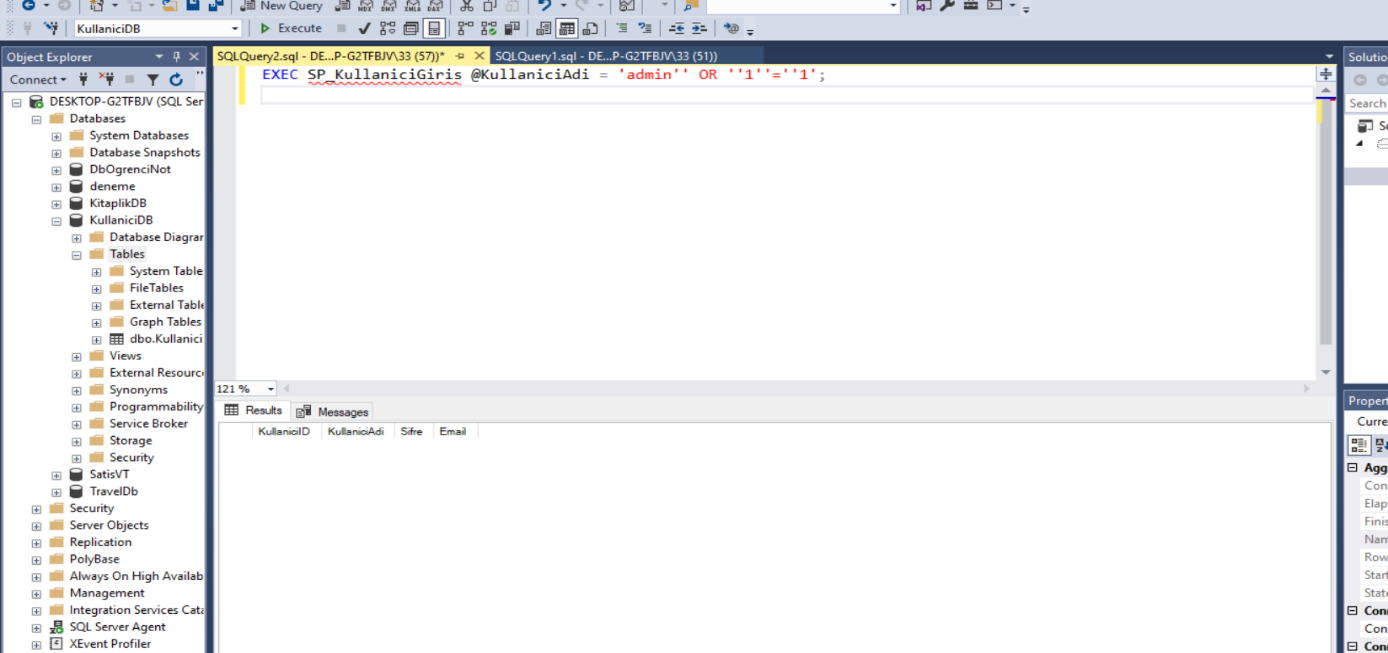
Burada ise yetkisiz sorgulama yapılıyor

**Adım 3 – SQL Injection Testi ve Korunma**

**Açıklama:**

SQL Injection saldırısı simüle edilerek, zararlı girişler test edilmiştir.  
Ardından güvenli sorgulama için Stored Procedure kullanılmıştır.



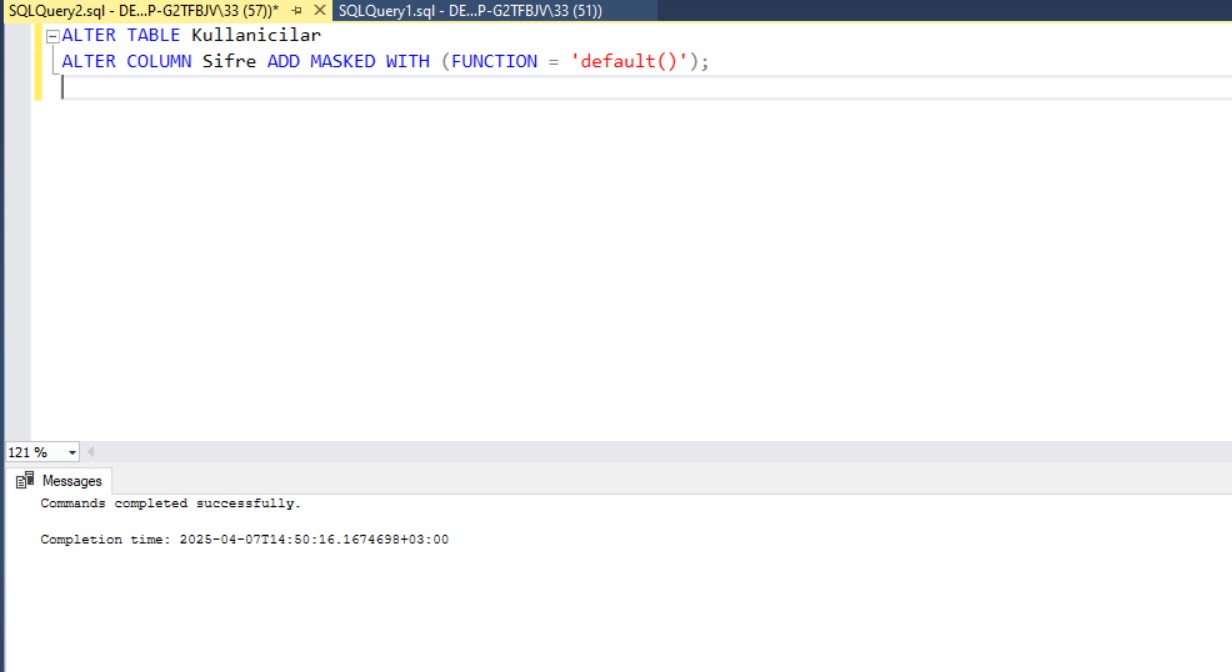


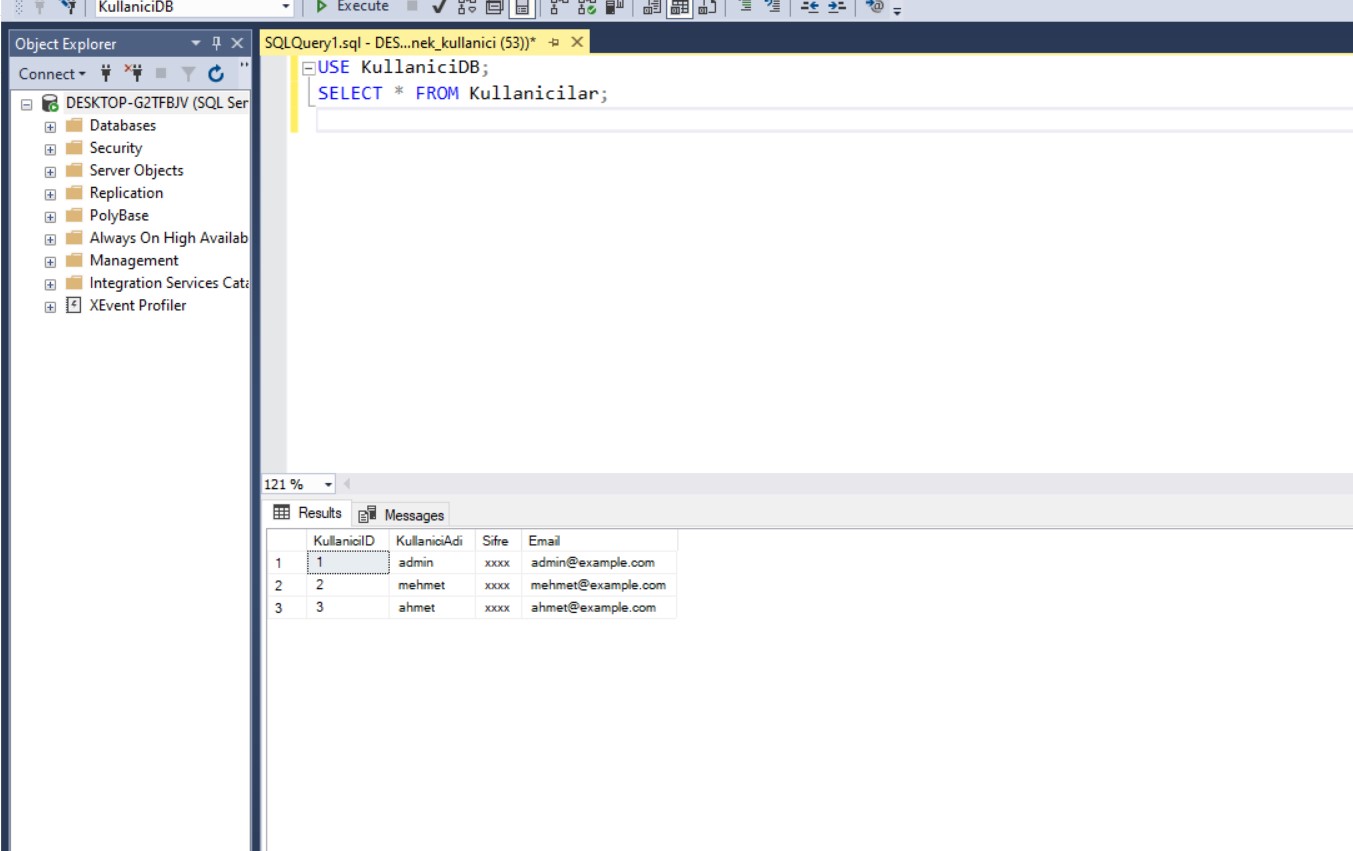
Burada injectiona karşı koruma gösterilmiştir.

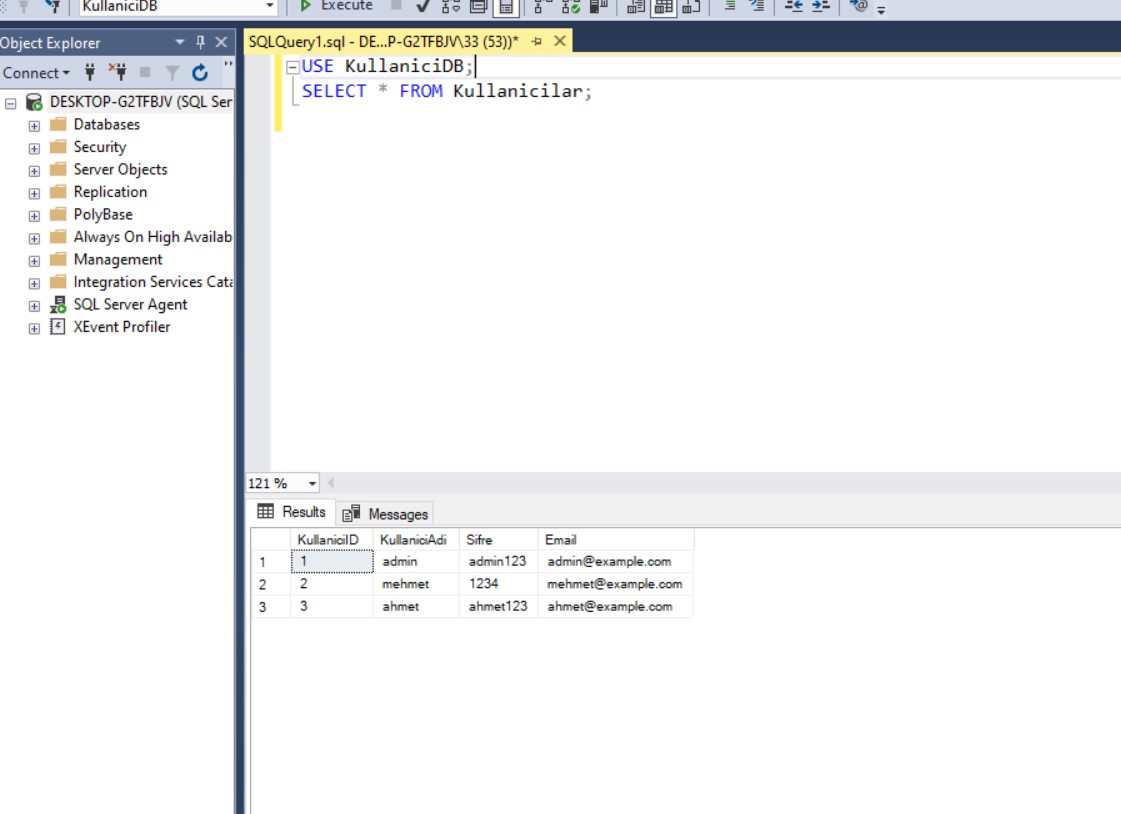
**Adım 4 – Verilerin Maskelemesi**

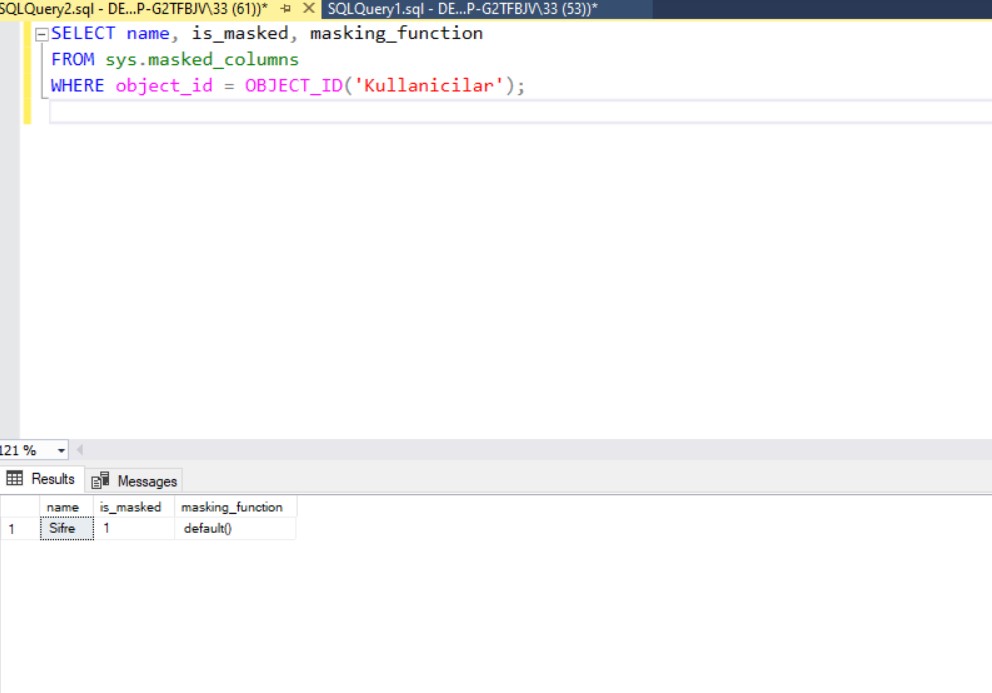
**Açıklama:**

Sifre kolonuna dinamik maskeleme uygulanarak yetkisiz kullanıcıların görememesi sağlanmıştır.





ornek\_kullanici ile bağlanıp Sifre kolonunun maskelenmiş haliYönetici ile aynı sorguda Sifre kolonunun görünmesi

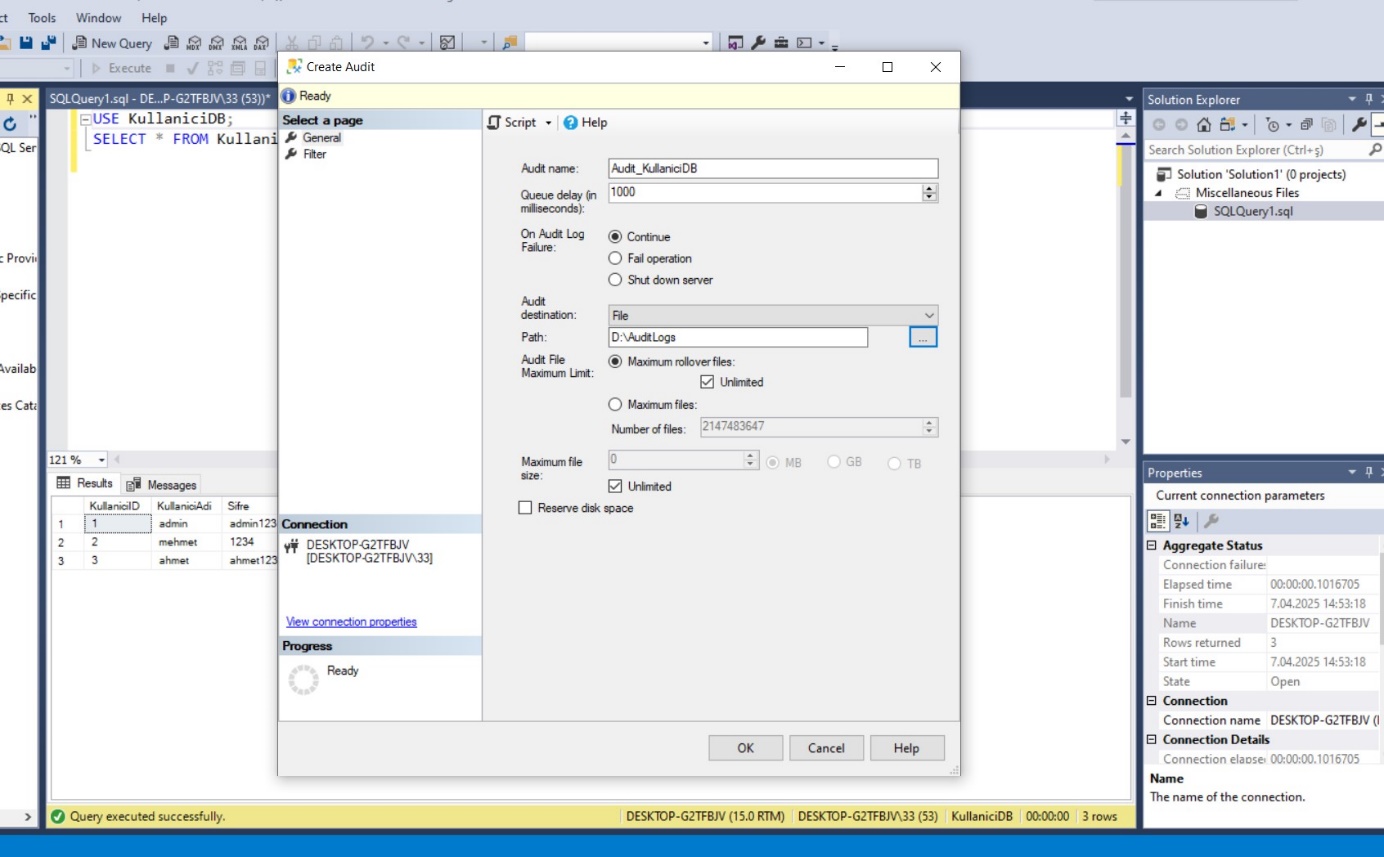


sys.masked\_columns ile kontrol ekranı

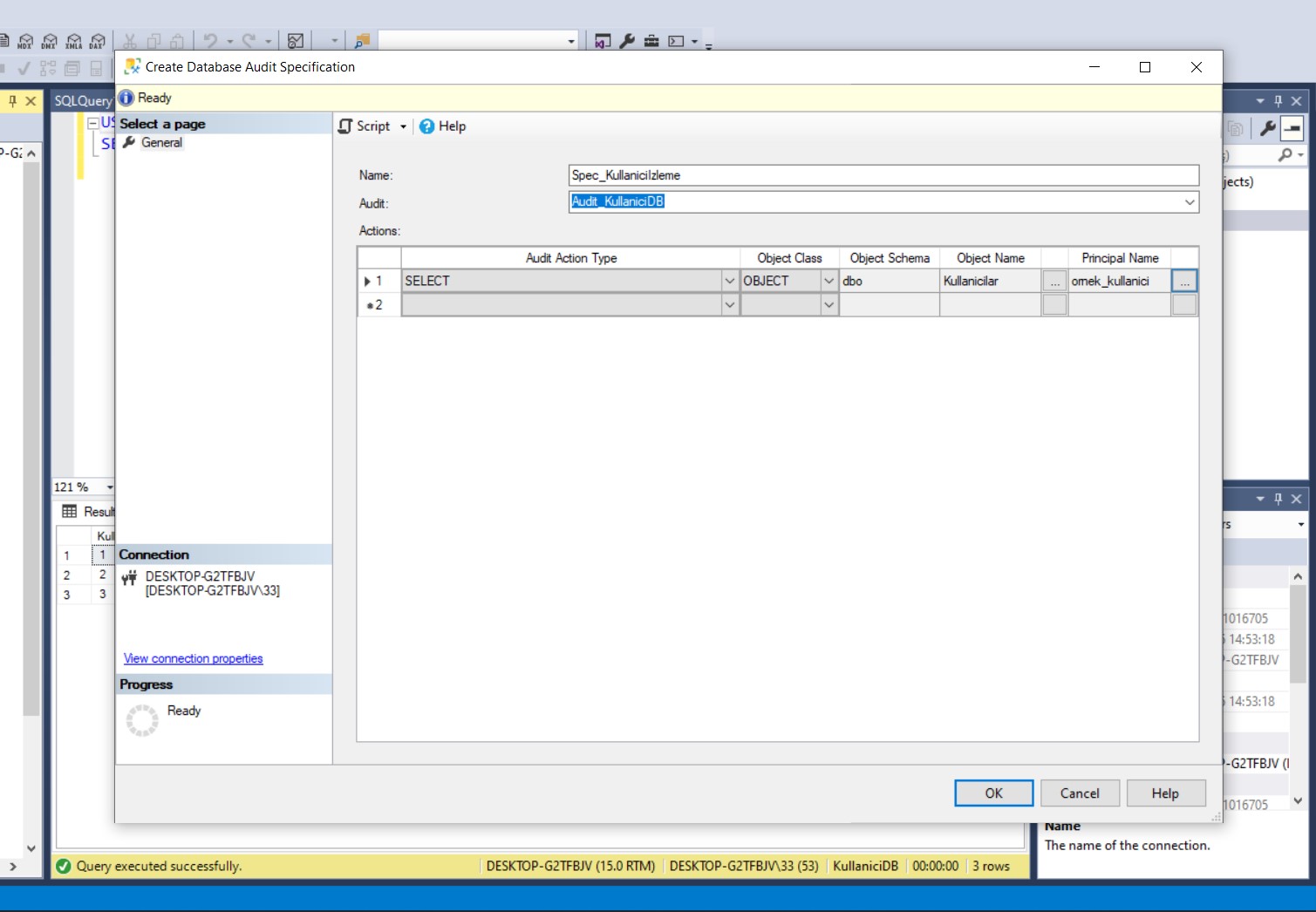
**Adım 5 – Audit: Kullanıcı İşlemlerini İzleme**

**Açıklama:**

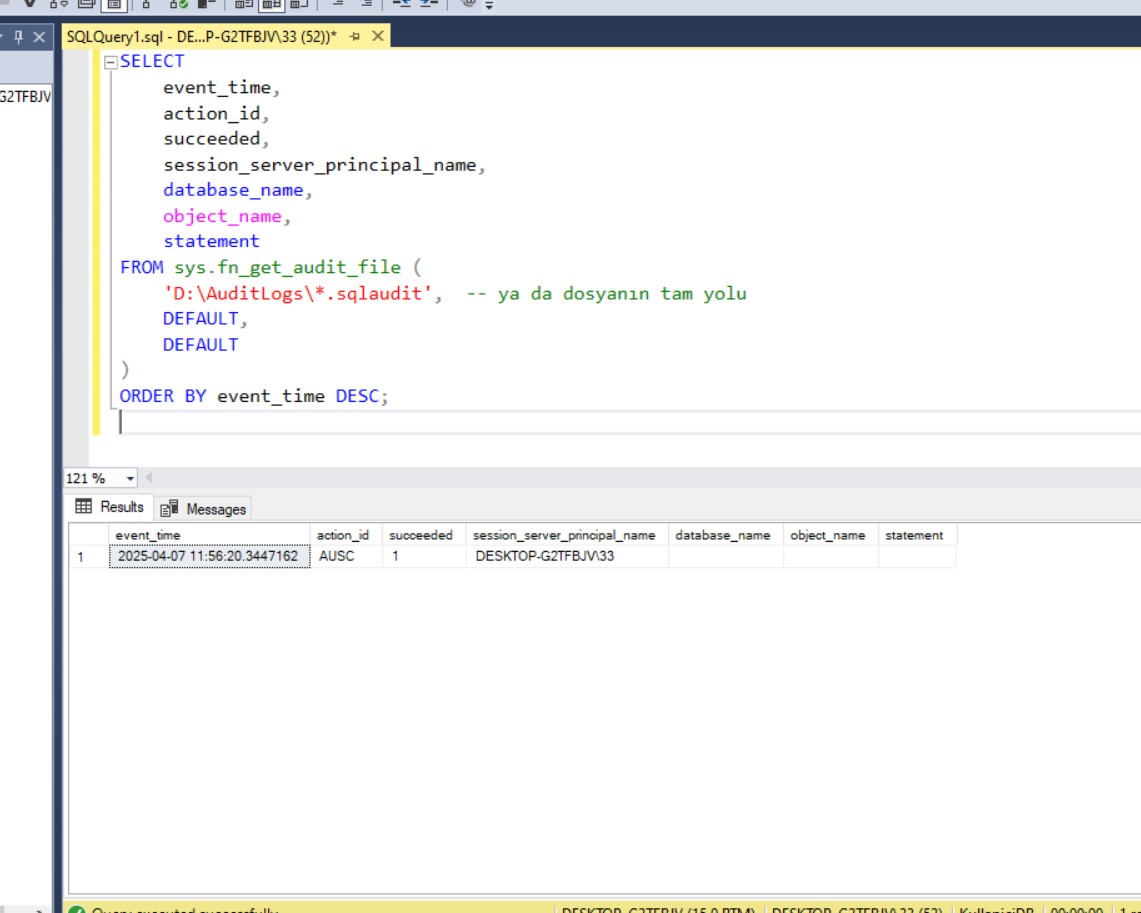
ornek\_kullanici tarafından yapılan SELECT sorguları SQL Server Audit sistemi ile izlenmiştir.

****

Audit\_KullaniciDB adlı audit tanımlandı



Database Audit Specification oluşturuldu

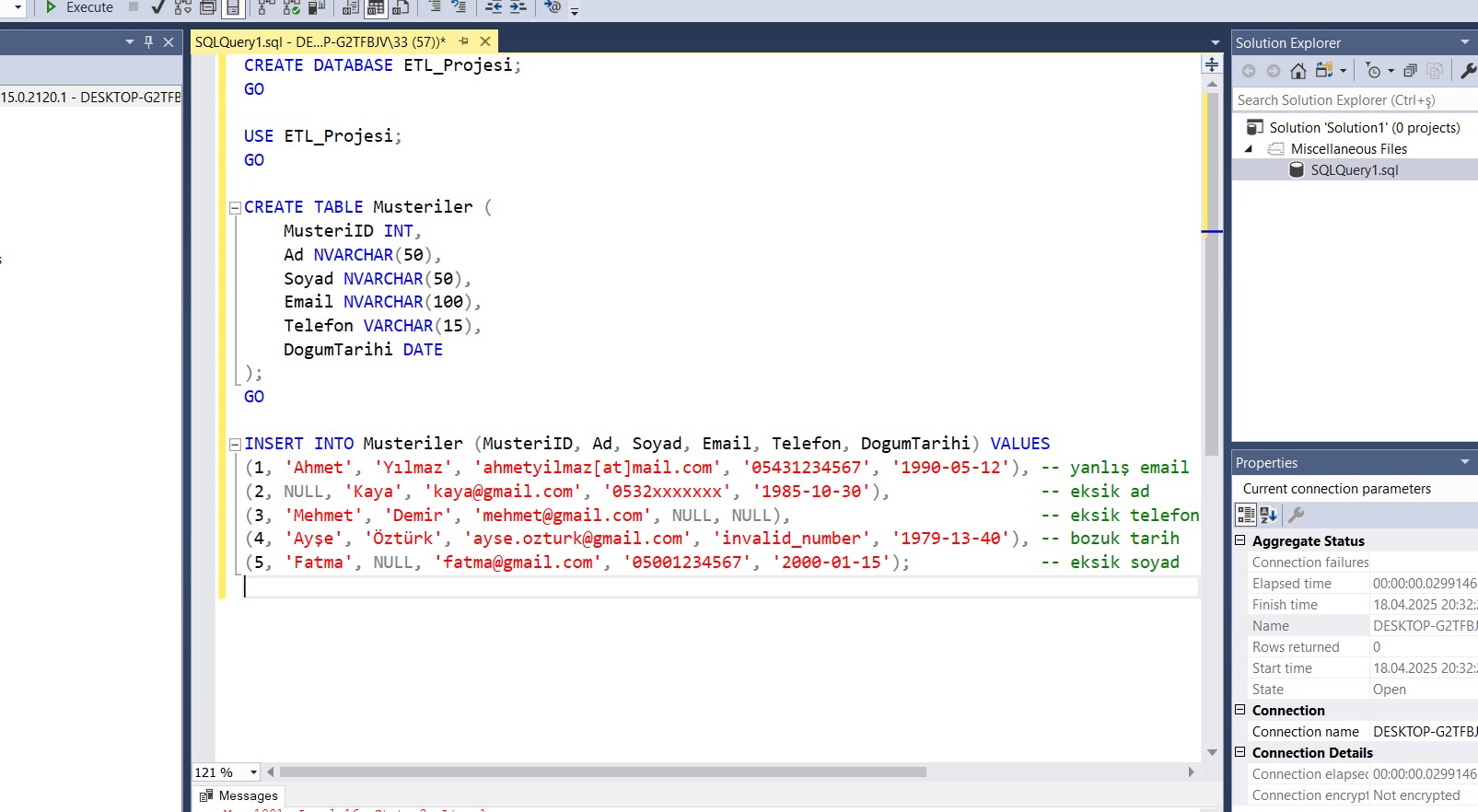


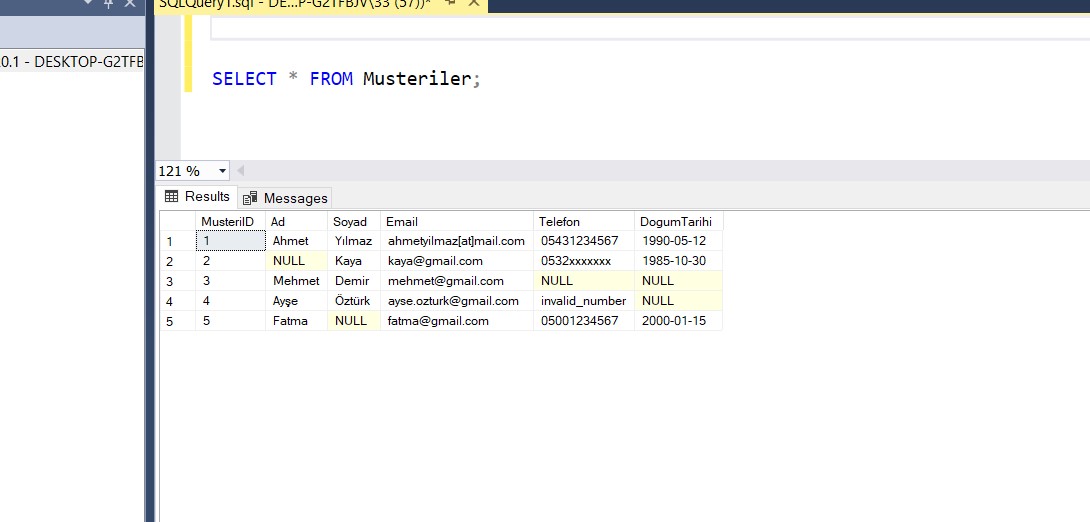
Kullanıcı sorgusundan sonra oluşan loglar

3) Veri Temizleme ve ETL Süreci Uygulaması

Adım 1: Hatalı Veri İçeren Kaynak Tabloyu Oluşturma

ETL\_Projesi adında yeni bir veritabanı oluşturulmuş ve Musteriler adında bir tablo tanımlanmıştır. Bu tabloya bilinçli olarak hatalı ve eksik veriler eklenmiştir.

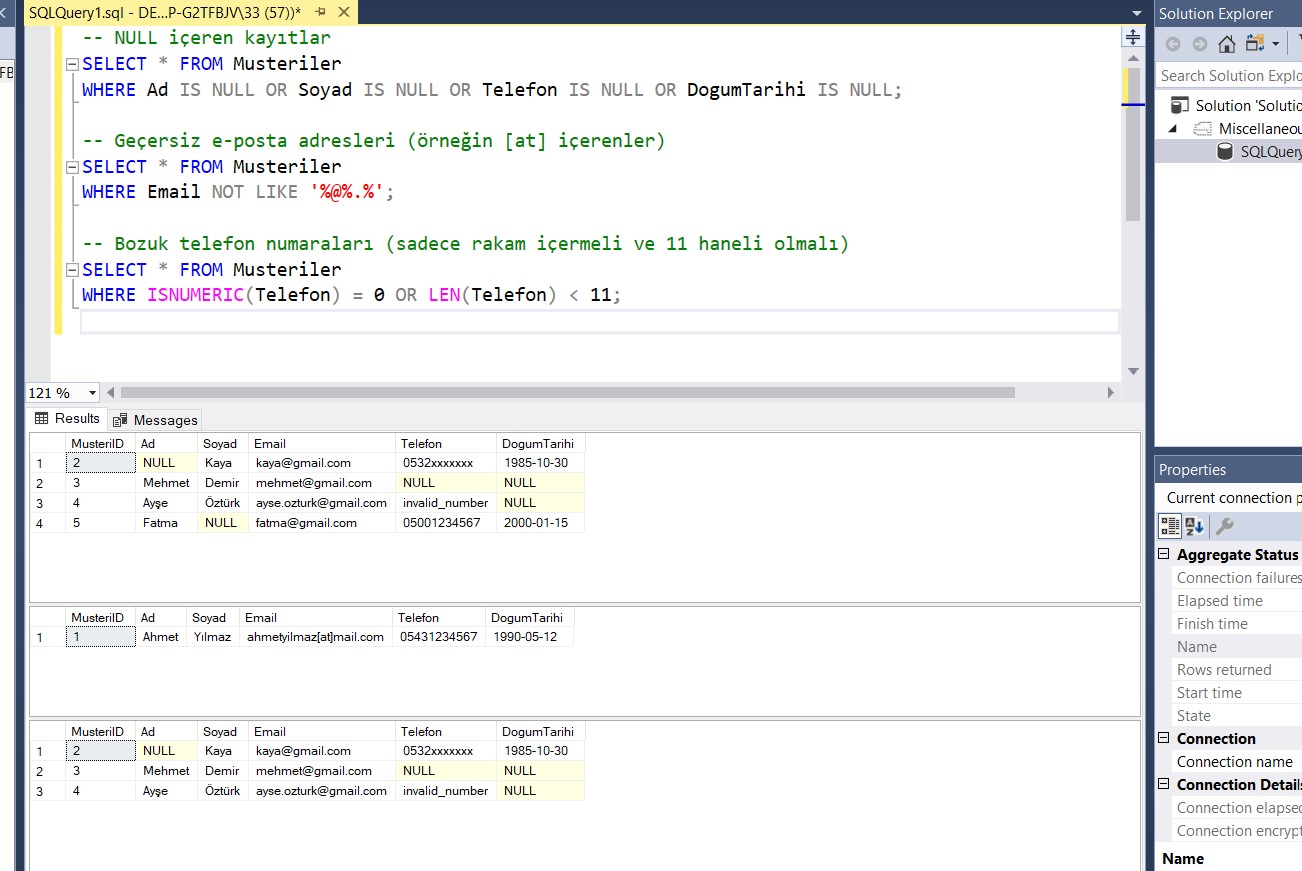


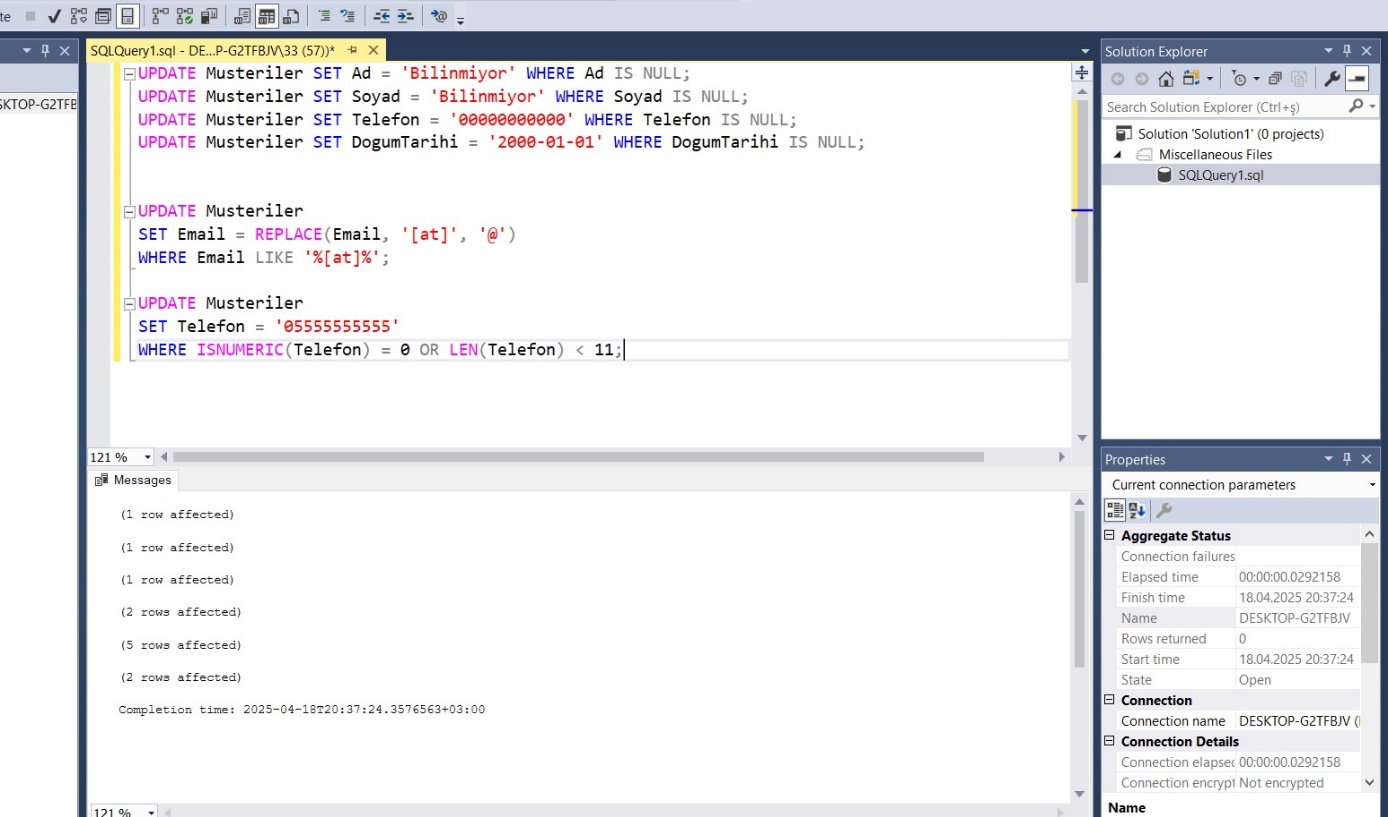


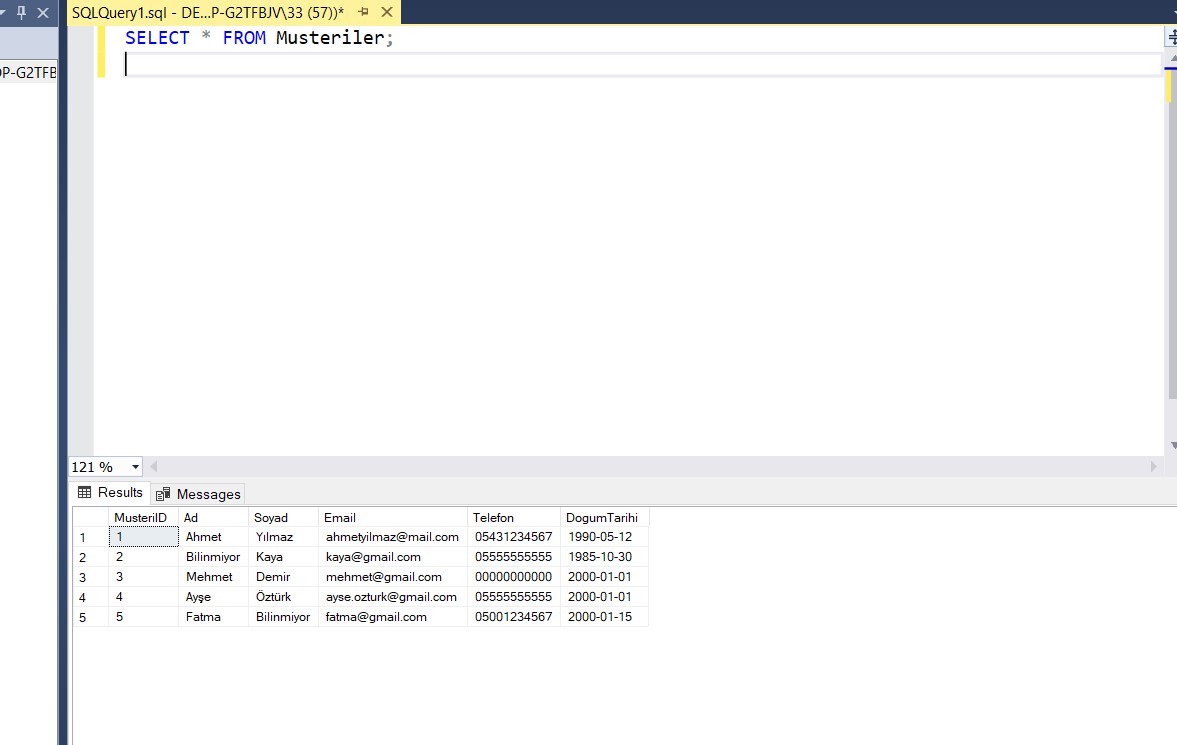
Adım 2: Veri Temizleme İşlemleri

Eksik veya hatalı veriler tespit edilip düzeltildi. Örneğin:

* NULL alanlar Bilinmiyor, 00000000000, 2000-01-01 gibi değerlerle güncellendi
* E-posta formatları düzeltildi ([at] → @)
* Bozuk telefon numaraları geçerli hale getirildi

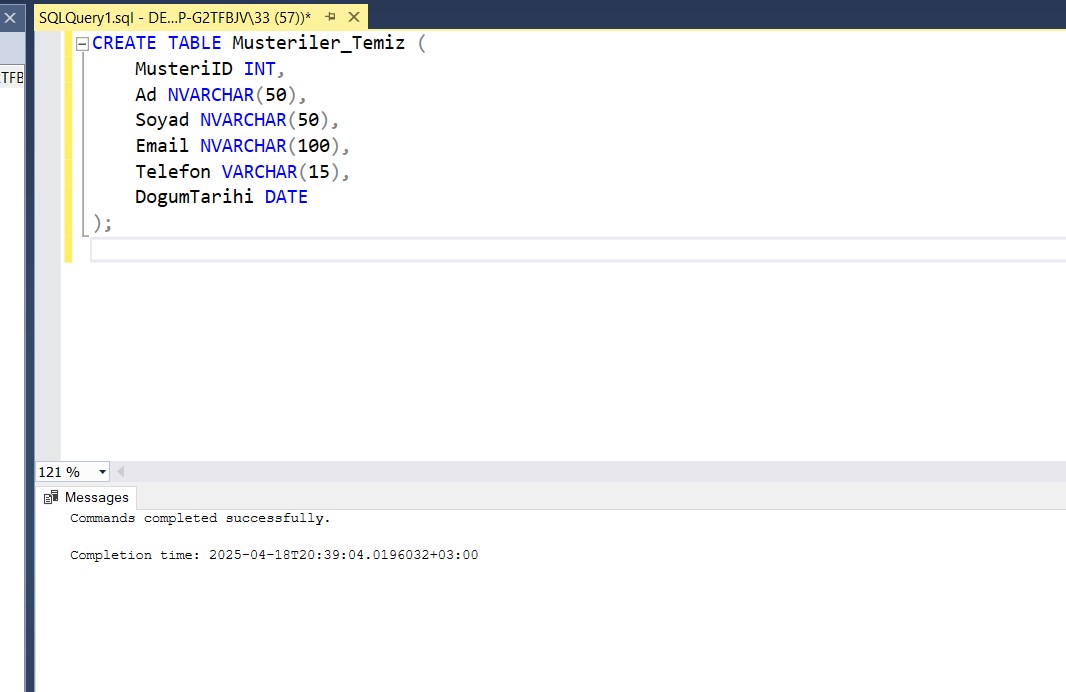




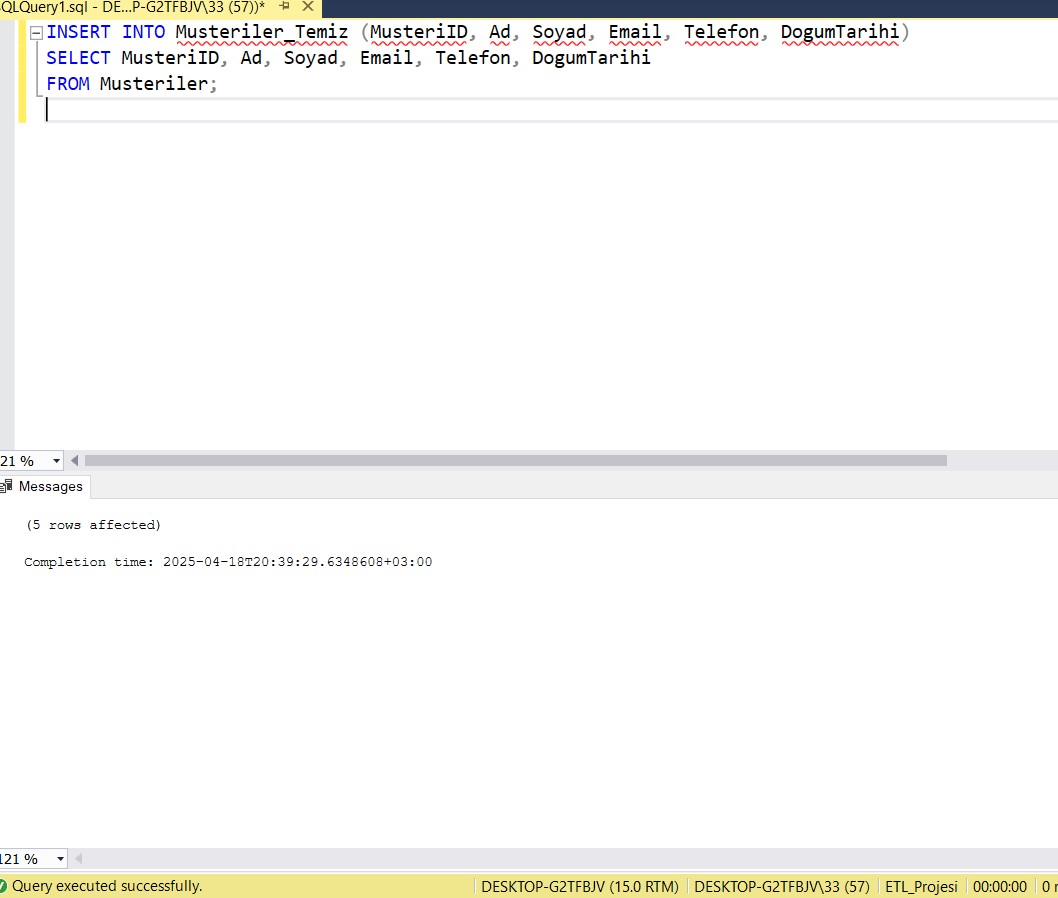


Adım 3: ETL - Temiz Veriyi Yeni Tabloya Aktarma (Load)

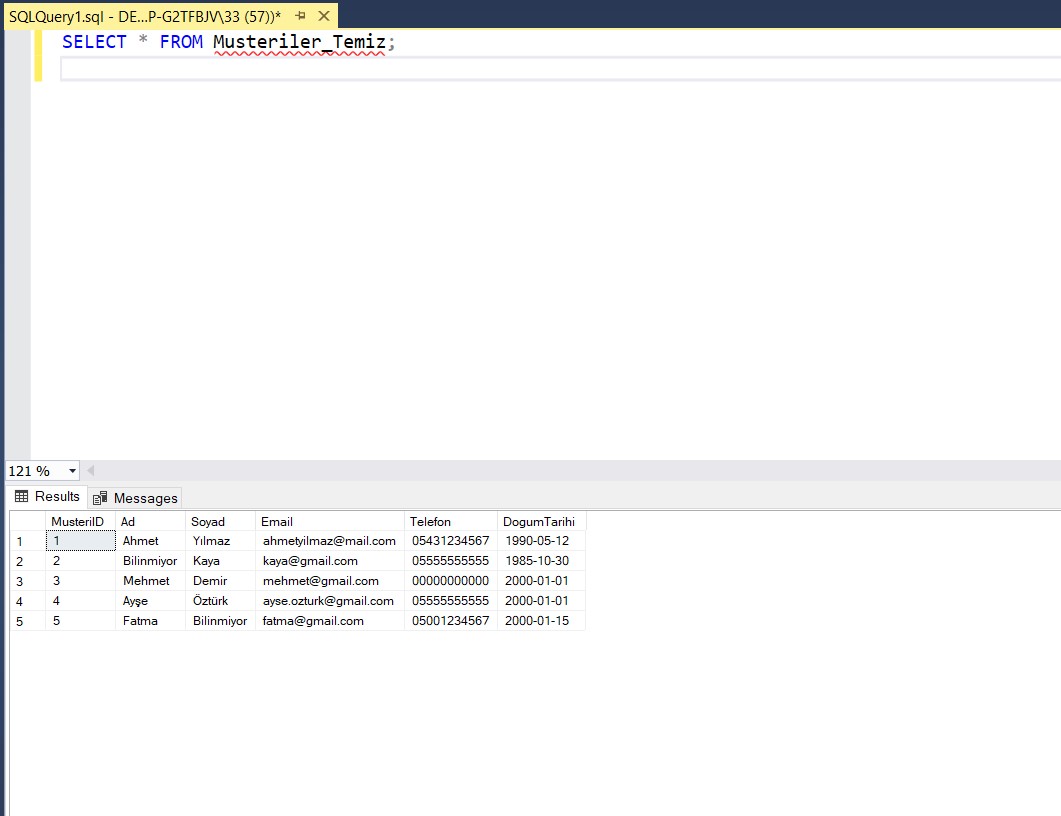
Temizlenmiş veriler Musteriler\_Temiz adlı yeni tabloya aktarılmıştır. Bu işlem ETL sürecinin "Load" adımını temsil eder.



Burada yeni bir tablo oluşturuldu.



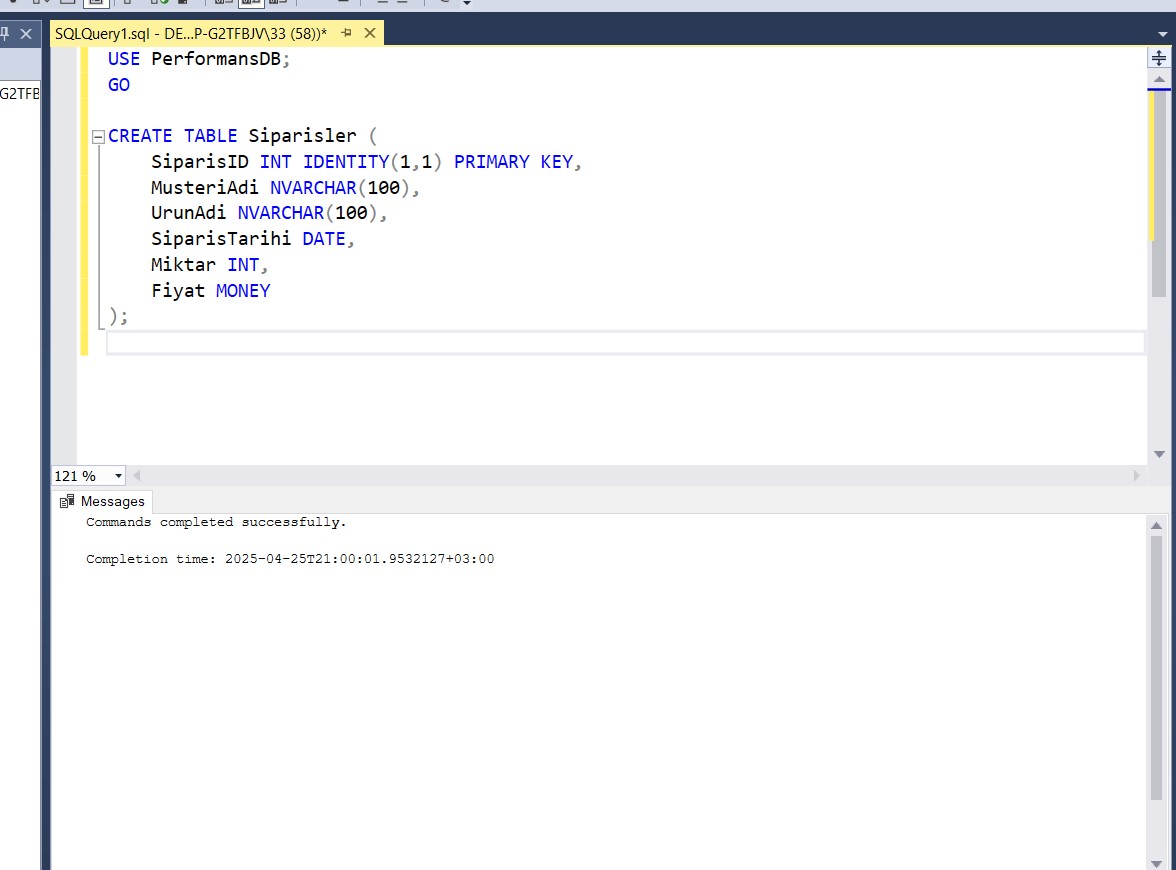
Temiz veriyi bu yeni tablomuza aktardık.



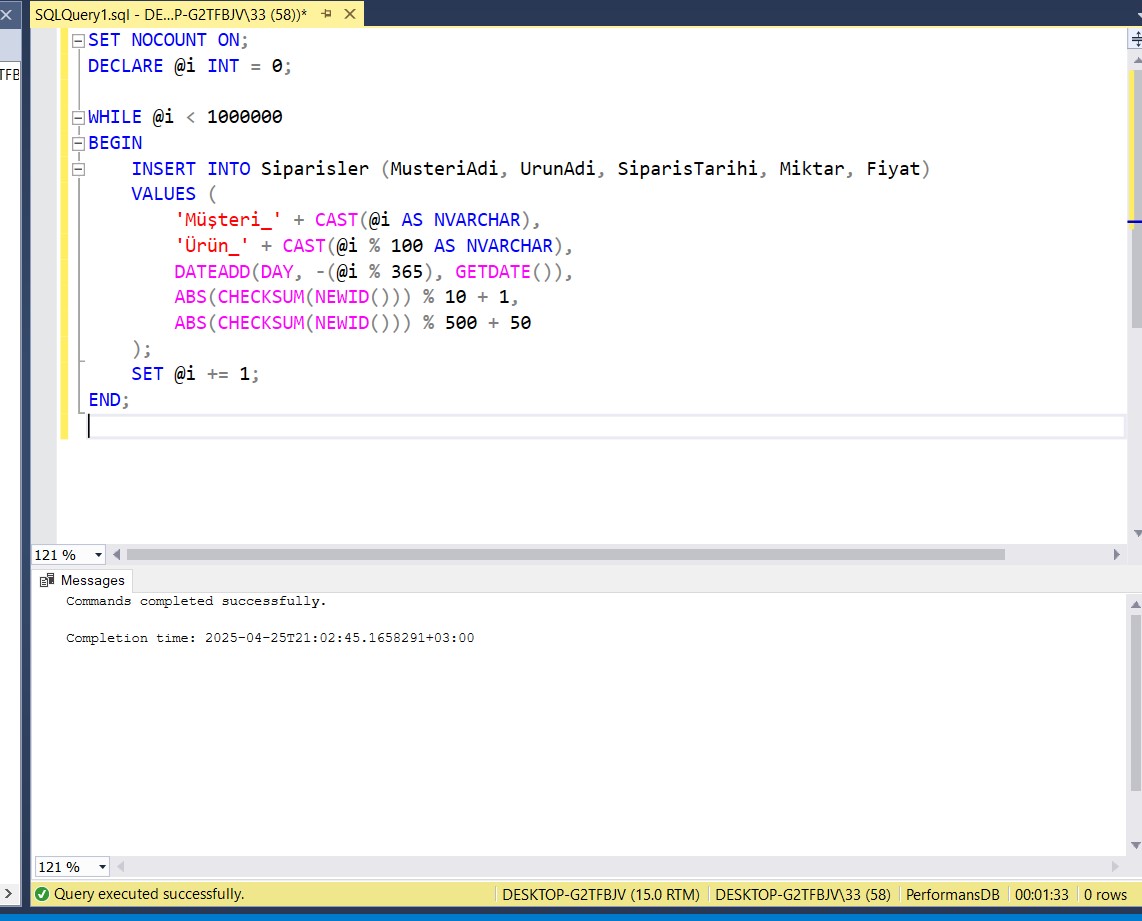
Bu da yeni tablomuz.

4) Veritabanı Performans Optimizasyonu ve İzleme

Adım 1 – Veritabanı ve Örnek Veri Oluşturma

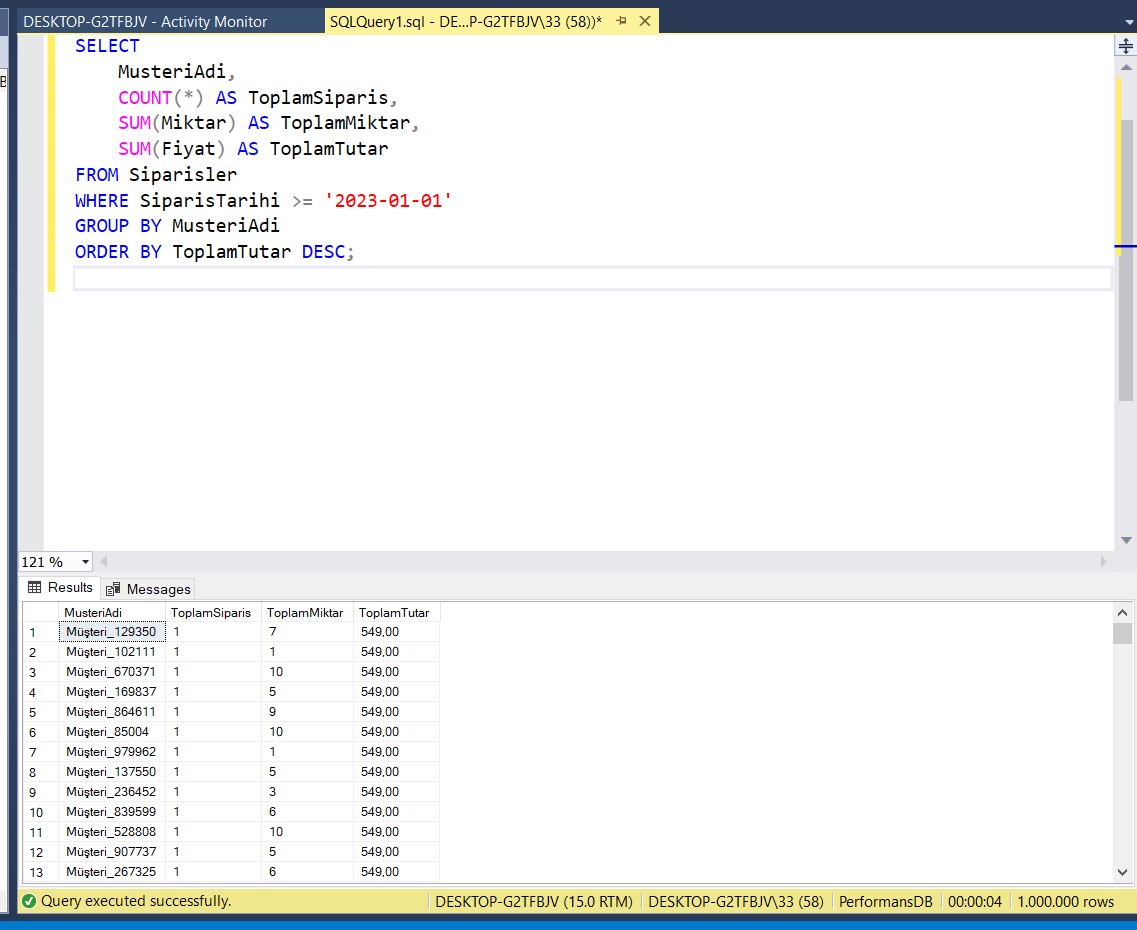


Performans testleri için kullanılacak Siparisler adında örnek bir tablo oluşturulmuştur. Bu tabloda müşteri adı, ürün adı, sipariş tarihi, miktar ve fiyat gibi alanlar yer almakta olup, her siparişin benzersiz bir kimliği olması için SiparisID alanı IDENTITY ile tanımlanmıştır.

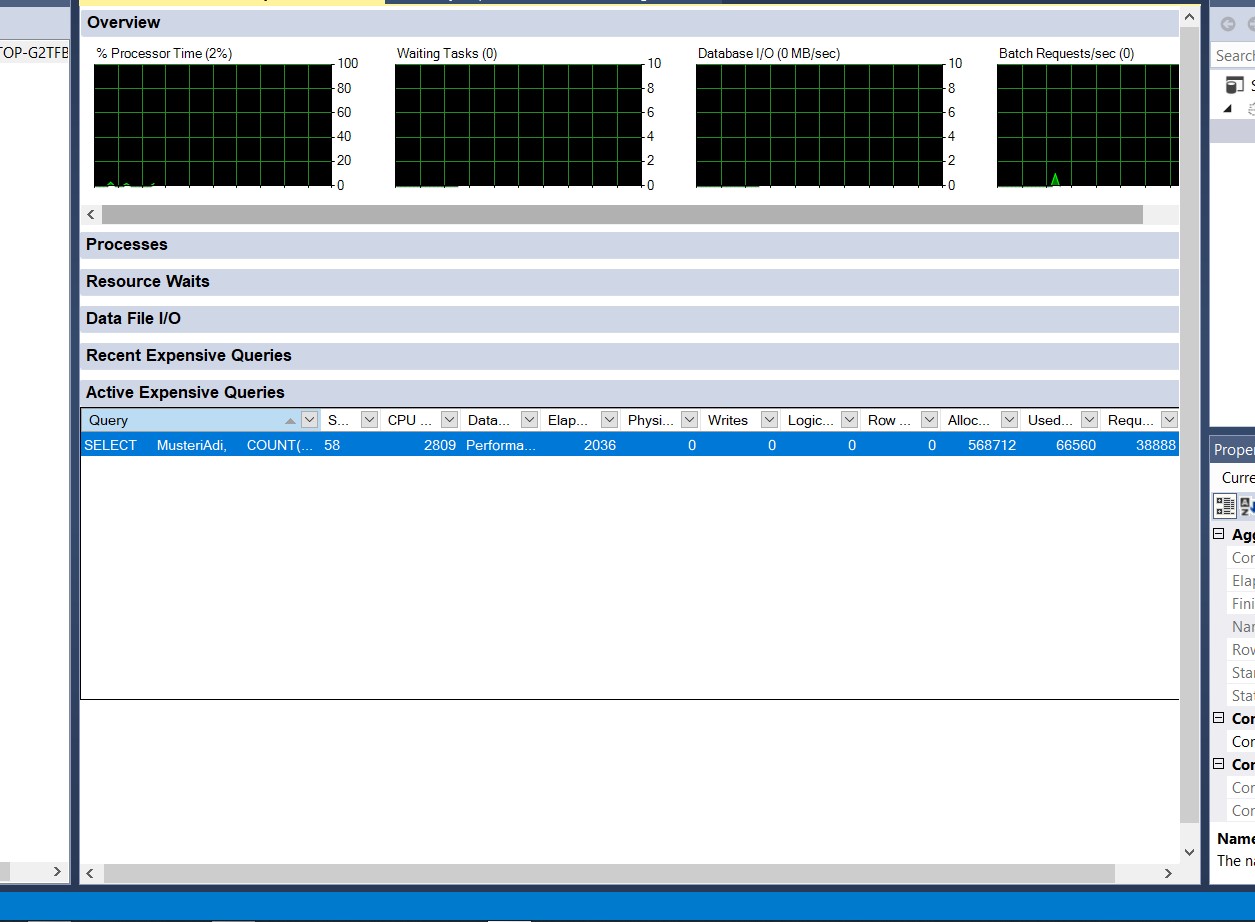


Performans testlerinin gerçekçi olması adına Siparisler tablosuna 1 milyon sahte veri girilmiştir. WHILE döngüsü ile rastgele müşteri ve ürün isimleri, geçmişe dönük rastgele tarihler ve miktar/fiyat kombinasyonları üretilmiştir. Bu işlem SQL Server üzerinde yüksek veri yoğunluğu oluşturarak ileride yapılacak performans analizleri için temel hazırlamıştır.

Adım 2: Ağır Sorgu ile Performans Yükü Oluşturma



Veritabanı performansını analiz edebilmek ve SQL Profiler, Activity Monitor gibi araçlarda gözlem yapabilmek için sistem üzerinde **yük oluşturacak kasıtlı bir ağır sorgu** çalıştırılmıştır. Bu sorgu 1 milyon satırlık Siparisler tablosu üzerinde GROUP BY ve ORDER BY işlemleri ile kaynak kullanımını artırmıştır.

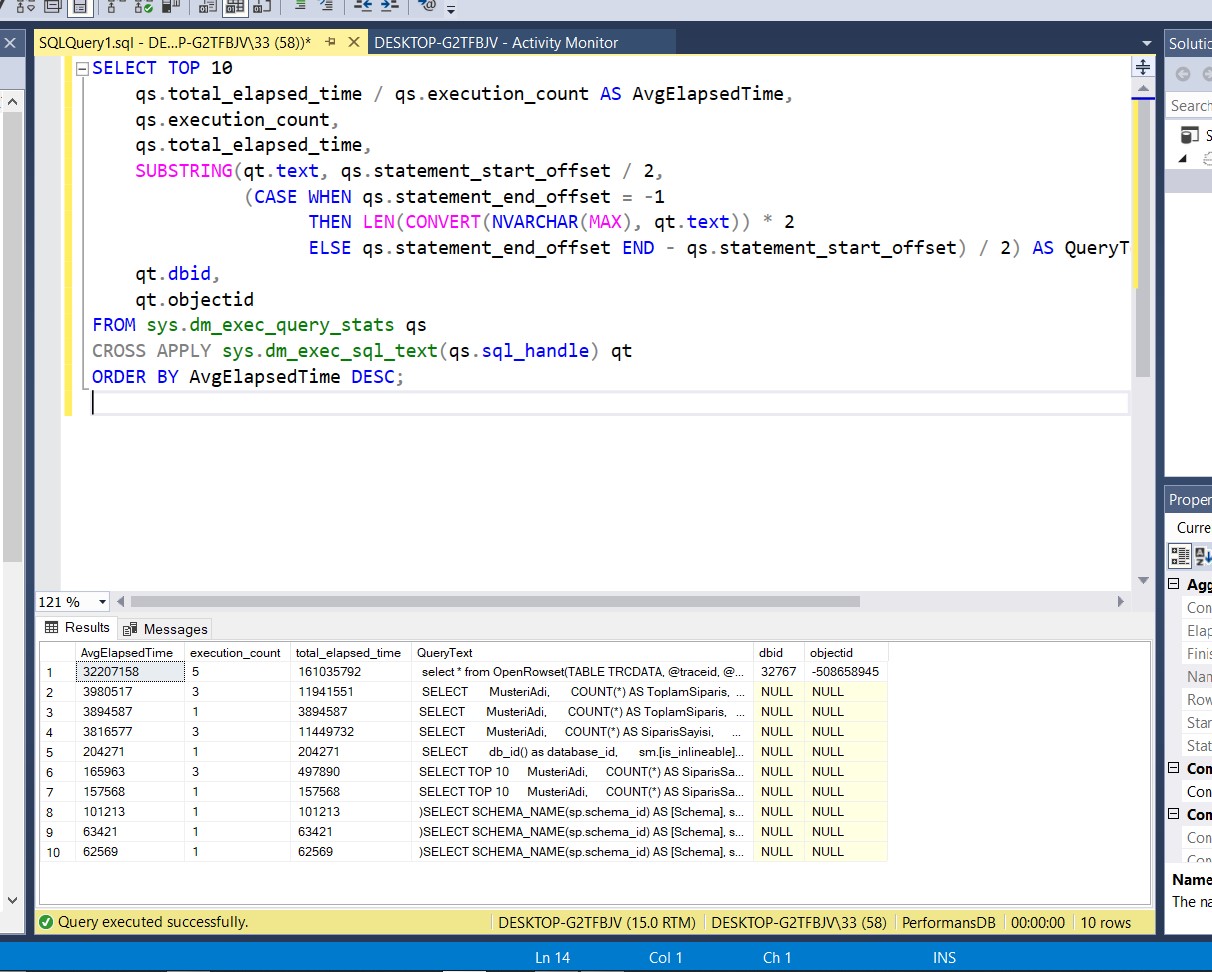


SQL Server'ın yerleşik aracı olan **Activity Monitor** kullanılarak, çalıştırılan ağır sorgunun sistem kaynakları üzerindeki etkisi gözlemlenmiştir.

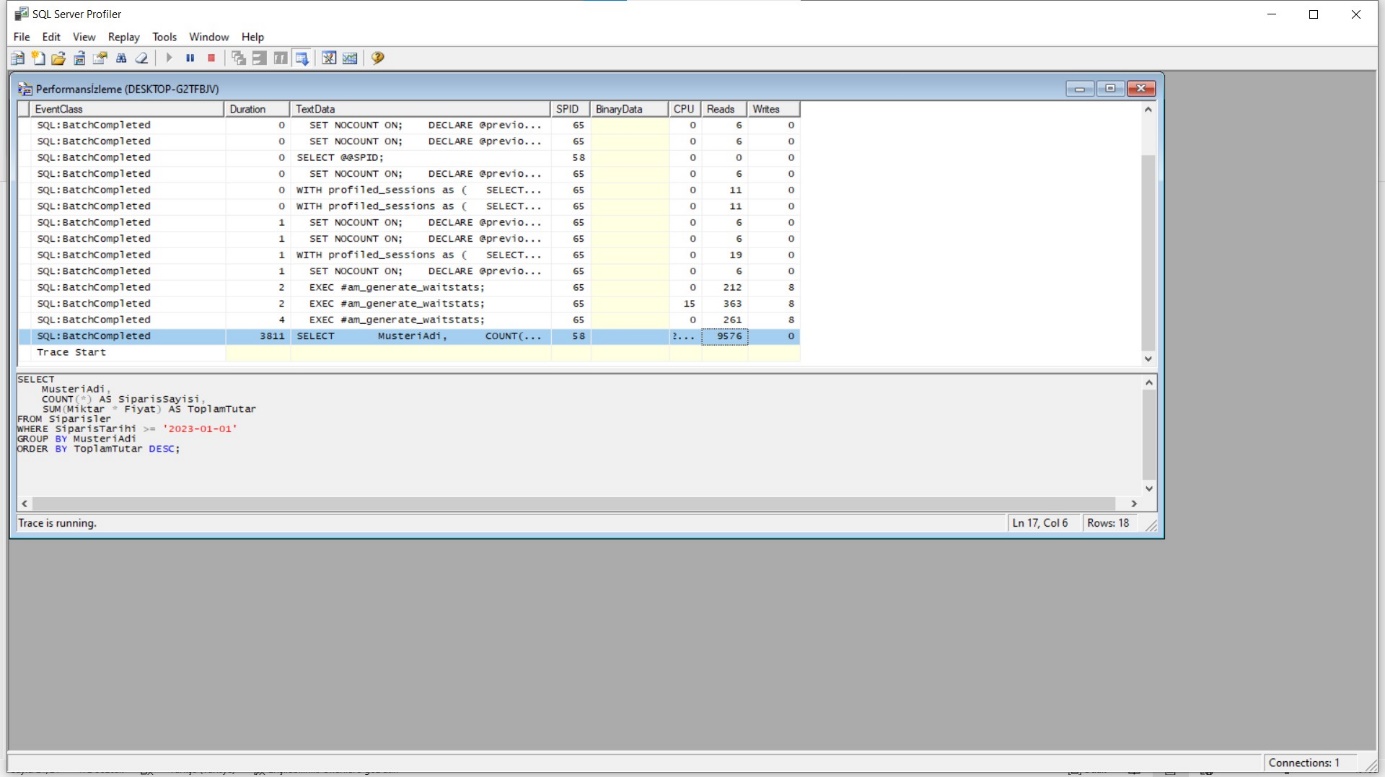
Öne çıkan metrikler:

* **CPU Süresi:** 2809 ms
* **Elapsed Time (Geçen Süre):** 2036 ms
* **Memory Used (KB):** 66560
* **Allocated Memory (KB):** 568712

Bu ekran, sistem üzerinde en çok kaynak tüketen sorguların görüntülenmesini sağlar ve performans darboğazlarını tespit etmekte önemli bir rol oynar.



SQL Server'daki sys.dm\_exec\_query\_stats DMV'si kullanılarak, geçmişte çalıştırılan sorgular arasından **en uzun sürede tamamlananlar (AvgElapsedTime)** belirlenmiştir. Bu analiz, hangi sorguların optimize edilmesi gerektiğine karar vermede önemli rol oynar.



SQL Server Profiler kullanılarak sistem üzerindeki sorguların gerçek zamanlı takibi yapılmıştır. Profiler sayesinde her bir sorgunun:

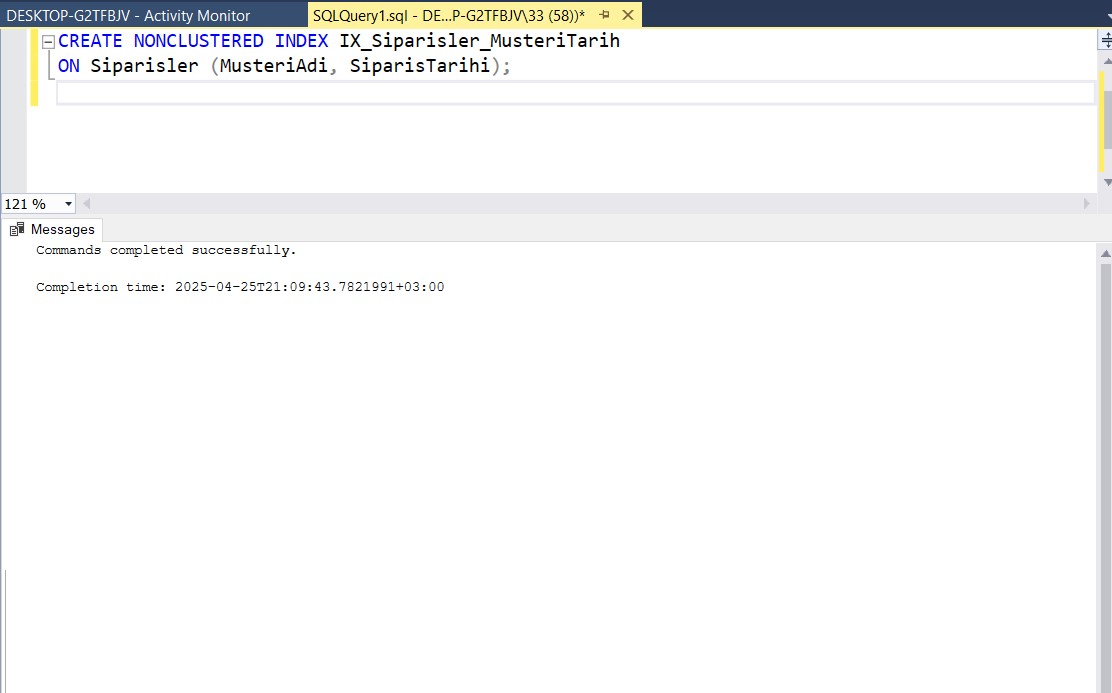
* **TextData (komut içeriği)**
* **CPU kullanımı**
* **Reads (veri okuma miktarı)**
* **Süre (Duration)**

gibi detaylı bilgileri toplanmıştır.

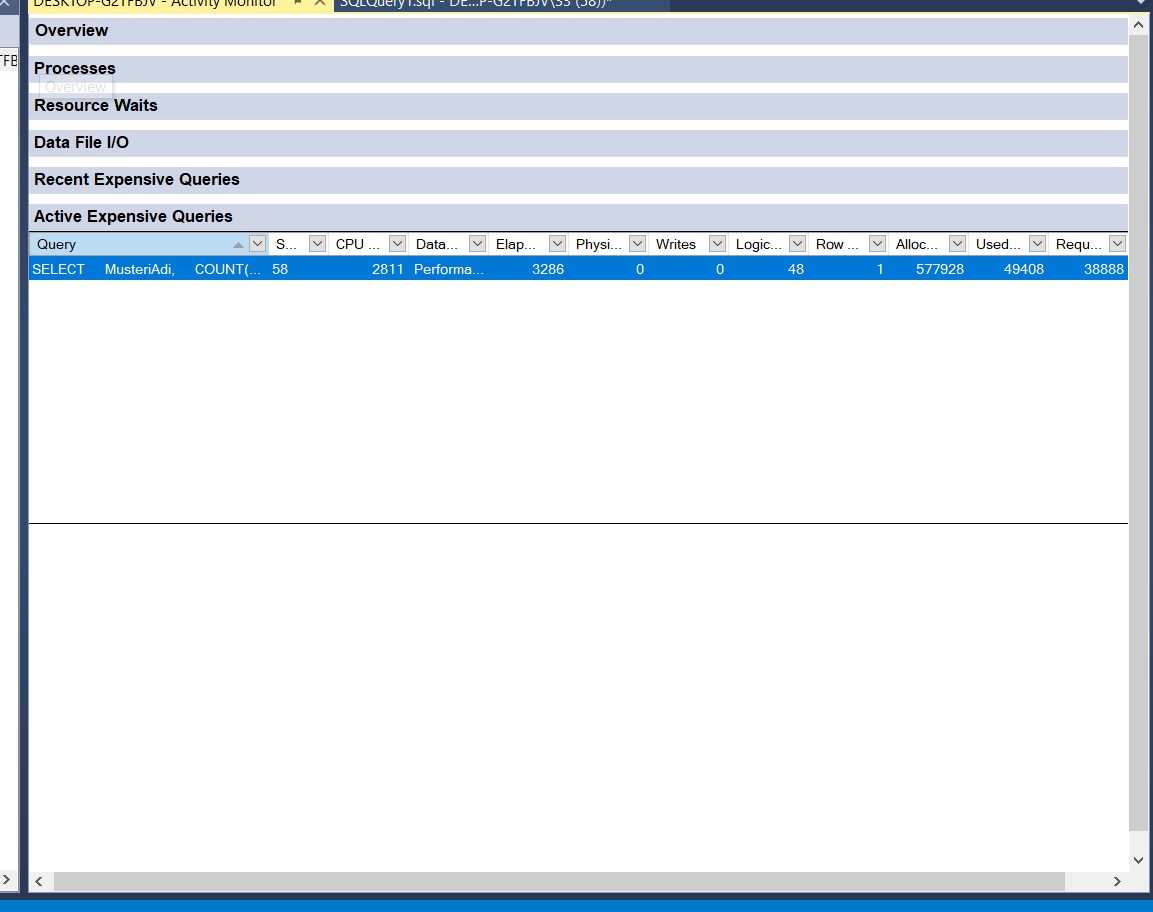
Gözlemler:

* **CPU:** 2000+
* **Reads:** 9576
* **Duration:** 3811 ms

Adım 3: Sorgu Performansını Artırmak İçin İndeks Kullanımı



Veri tablosunda en çok filtrelenen ve gruplanan sütunlara uygun bir şekilde indeks tanımlanmıştır.  
MusteriAdi ve SiparisTarihi alanlarına oluşturulan bileşik indeks ile sorguların erişim ve filtreleme süresi azaltılmıştır.



İndeks oluşturulduktan sonra aynı sorgu tekrar çalıştırılmış ve Activity Monitor aracılığıyla sistem üzerindeki etkisi gözlemlenmiştir.  
Aşağıdaki karşılaştırmada indeks öncesi ve sonrası durum kıyaslanmıştır:

| **Metrik** | **İndeks Öncesi** | **İndeks Sonrası** |
| --- | --- | --- |
| **CPU (ms)** | 2809 | 2811 |
| **Elapsed Time (ms)** | 2036 | 3286 |
| **Used Memory (KB)** | 66560 | 49408 |
| **Logical Reads** | 0 | 48 |
|  |  |  |

 CPU süresi benzer kalmış olsa da,

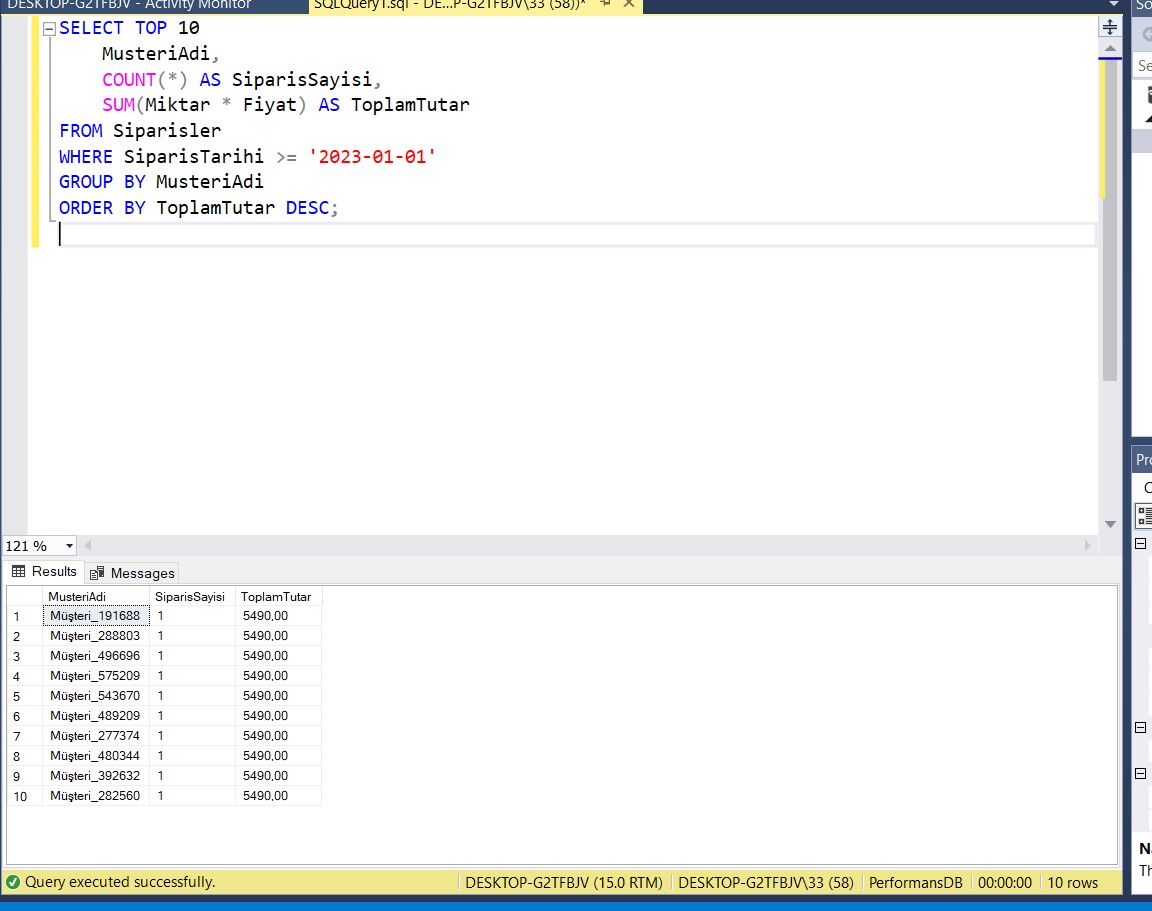
 **Bellek kullanımı düşmüş** ve

 **Okuma işlemleri (logical reads)** artmıştır → bu da indeksin devreye girdiğini gösterir.

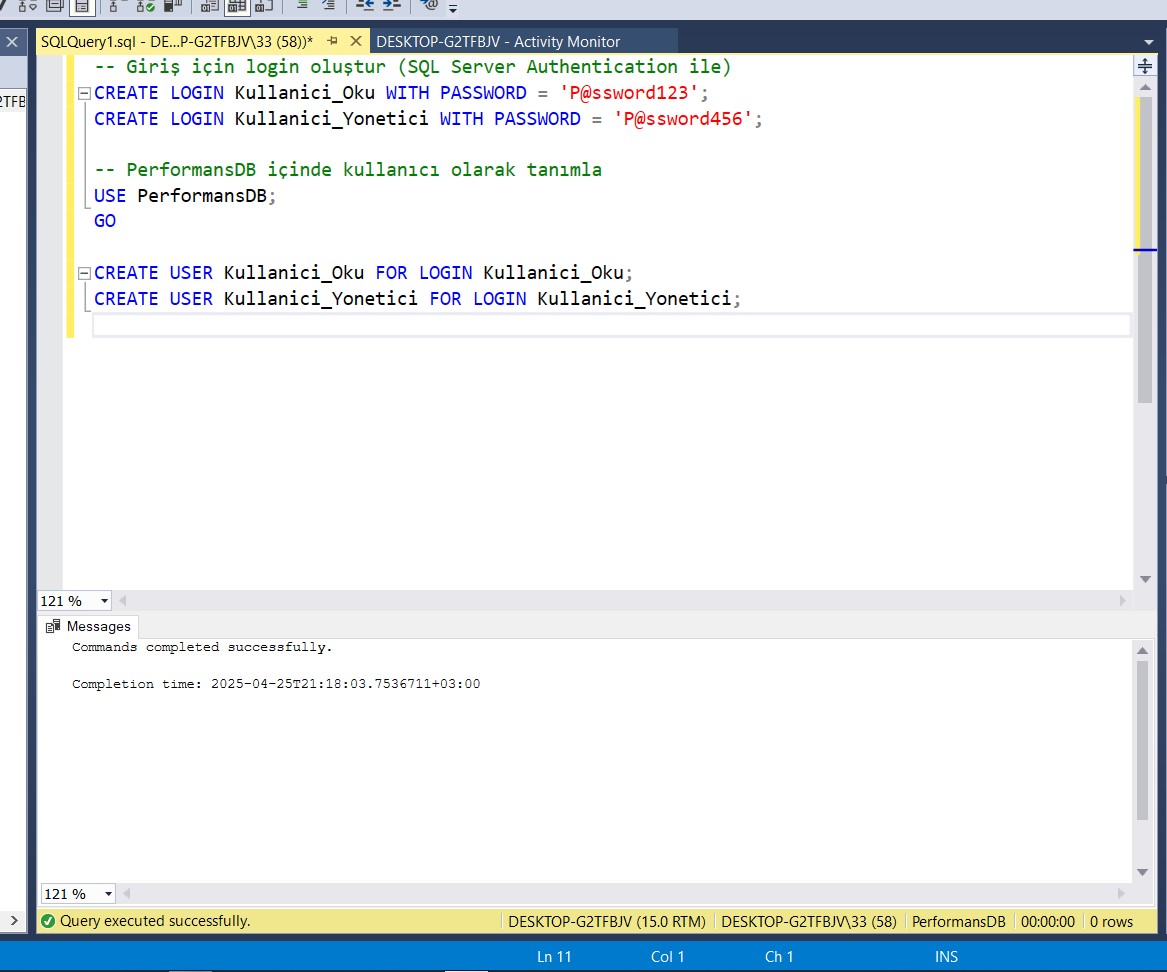
Adım 4: Optimize Edilmiş Sorgu ile Veri Çekme

Veritabanı performansını artırmak amacıyla optimize edilmiş sorgular kullanılmıştır. Bu örnekte:

* TOP 10 ifadesiyle sadece en çok tutar yapan ilk 10 müşteri çekilmiştir.
* WHERE filtresi doğrudan tarih karşılaştırmasıyla yazılmıştır (fonksiyon kullanılmadan).
* GROUP BY ve ORDER BY kombinasyonu ile analiz yapılmıştır.



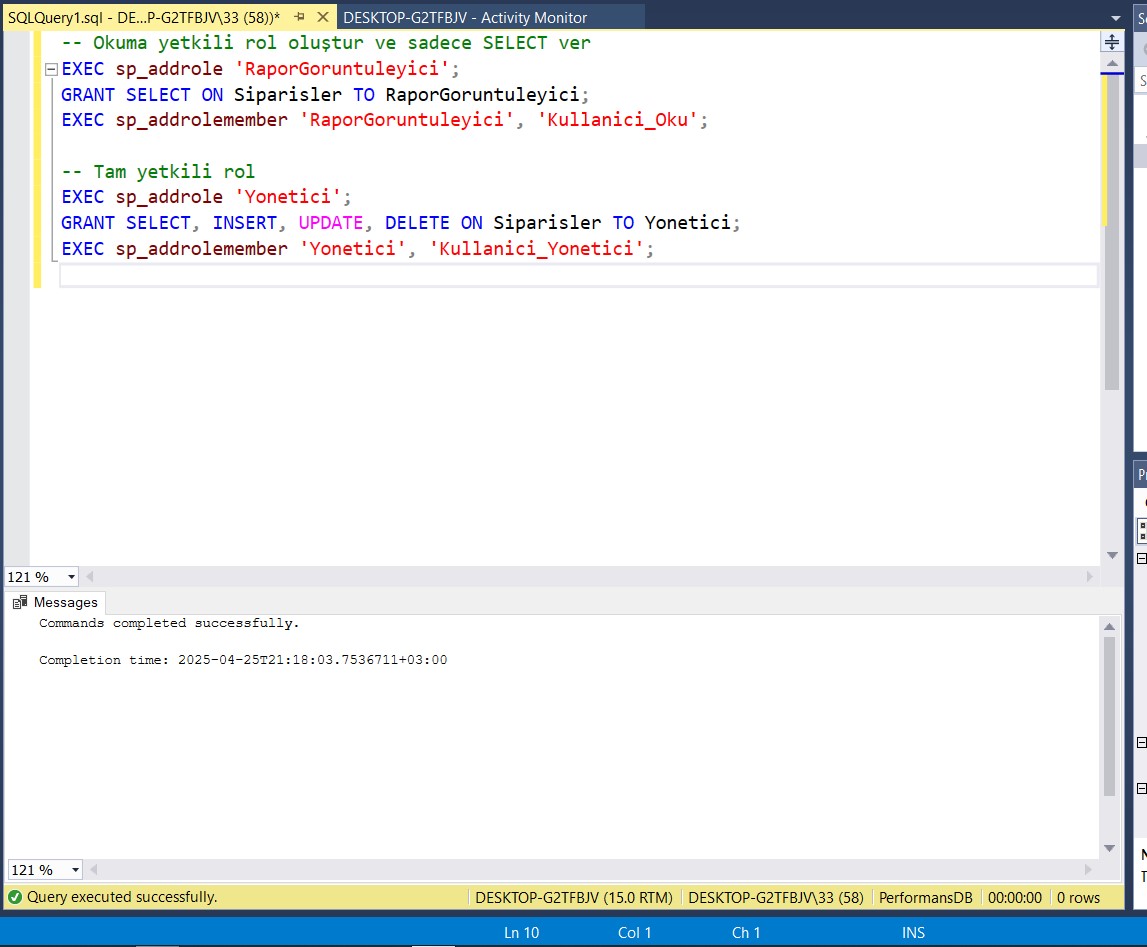
Adım 5: Veri Yöneticisi Rolleri



Veritabanı güvenliğini sağlamak amacıyla iki farklı kullanıcı tanımlanmıştır:

* Kullanici\_Oku: Sadece sorgu (SELECT) yetkisine sahip olacak kullanıcı
* Kullanici\_Yonetici: Sorgu çalıştırma yanında INSERT, UPDATE, DELETE gibi işlemleri de yapabilecek tam yetkili kullanıcı

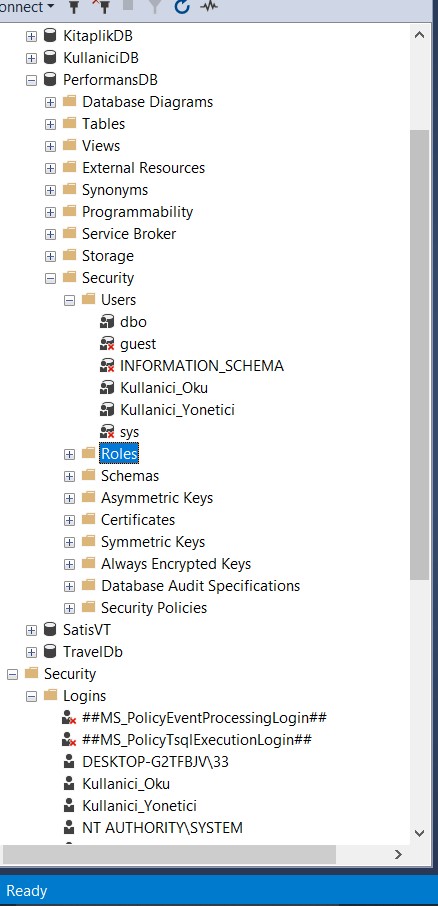
İlk adım olarak SQL Server Authentication ile giriş bilgileri oluşturulmuş, ardından bu log-in'ler **PerformansDB** veritabanında kullanıcı olarak tanımlanmıştır.



Veritabanı güvenliğini güçlendirmek ve kullanıcıların sadece ihtiyaç duyduğu verilere erişmesini sağlamak için **rol tabanlı erişim kontrolü** uygulanmıştır.

**➤ Oluşturulan Roller:**

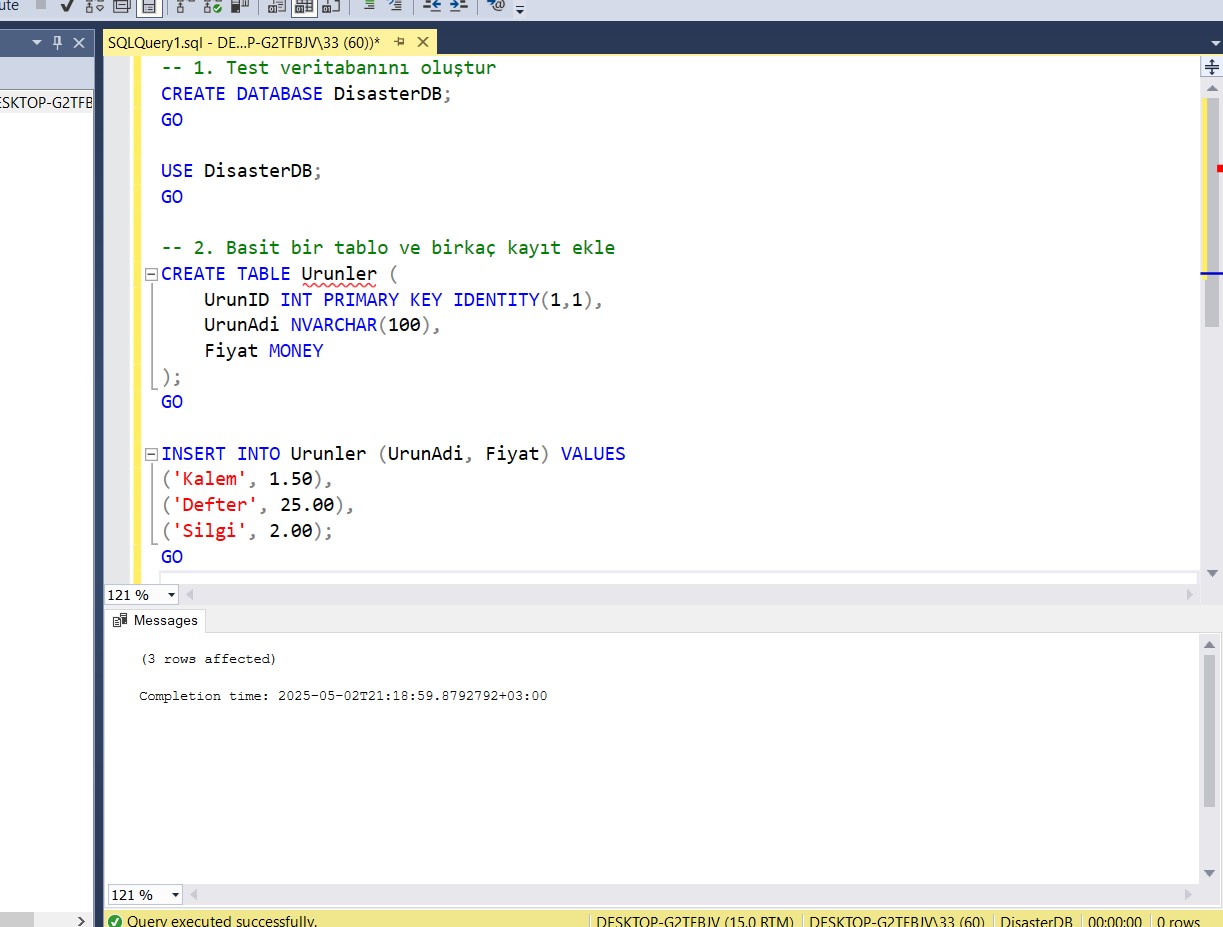
* RaporGoruntuleyici: Sadece SELECT yetkisi olan kullanıcılar (okuma yetkisi)
* Yonetici: SELECT, INSERT, UPDATE, DELETE gibi tüm yetkilere sahip olan yöneticiler



Bu görselde hem Users, hem Roles, hem de Logins bölümleri açıkça görülüyor, bu da **rol bazlı kullanıcı yönetiminin eksiksiz yapıldığını** gösteriyor.

5) Veritabanı Yedekleme ve Felaketten Kurtarma Planı

Adım 0 – Test Veritabanı ve Örnek Veri Oluşturma



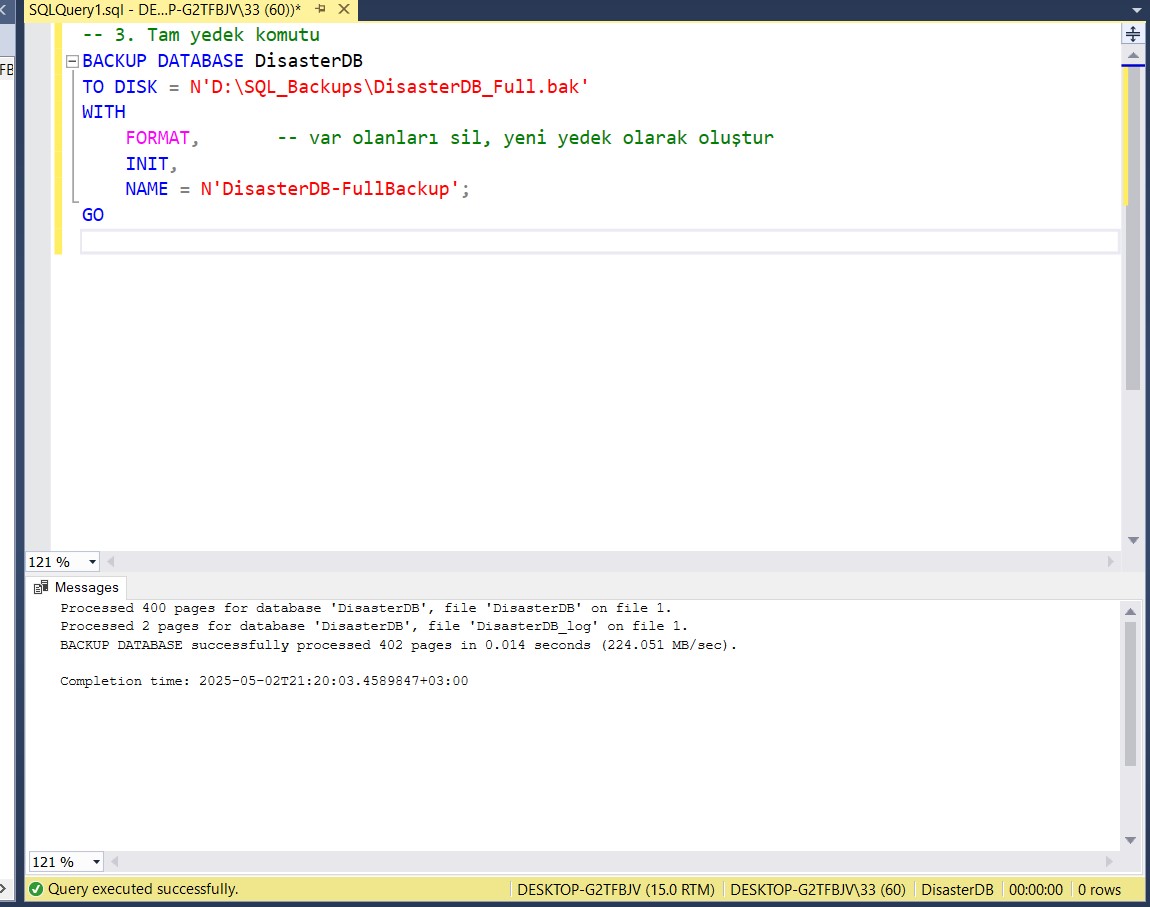
 CREATE DATABASE DisasterDB; ile felaket senaryosunu simüle edeceğimiz “DisasterDB” adlı boş veritabanını oluşturduk.

 USE DisasterDB; ile komutların bu veritabanı üzerinde çalışmasını sağladık.

 CREATE TABLE Urunler (…) ile içinde UrunID, UrunAdi ve Fiyat sütunları olan bir tablo tanımladık.

 INSERT INTO Urunler (…) VALUES (…) sorgusuyla tablomuza üç satır (“Kalem”, “Defter”, “Silgi”) ekleyerek test verisini yerleştirdik.

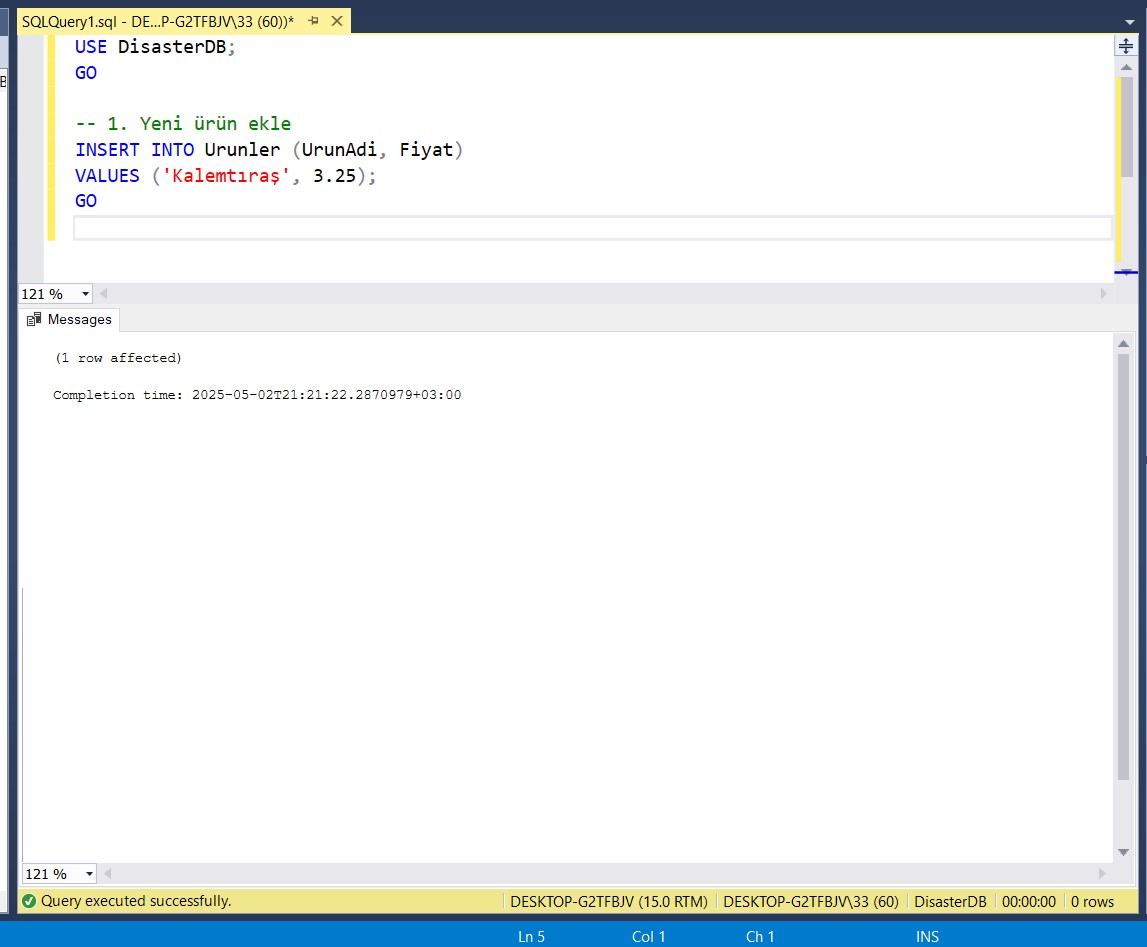
Adım 1 – Full Yedek Oluşturma



BACKUP DATABASE DisasterDB TO DISK = N'D:\SQL\_Backups\DisasterDB\_Full.bak' WITH FORMAT, INIT, NAME = N'DisasterDB-FullBackup';  
komutuyla “DisasterDB” veritabanının tam yedeğini aldık.

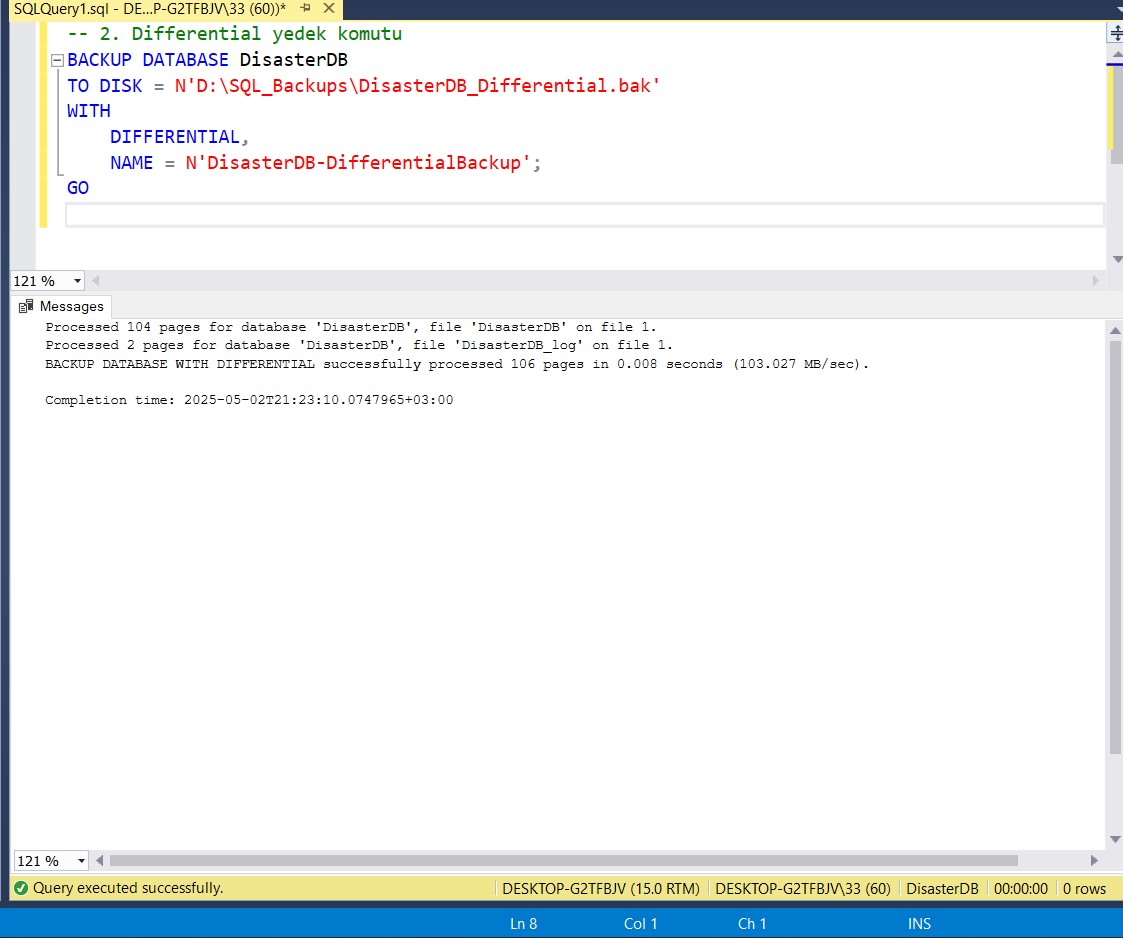
* FORMAT, INIT: Varolan yedek dosyasını silip yenisini oluşturur.
* NAME: Yedeğe açıklayıcı bir etiket atar.

Adım 2 – Transaction Log Yedeği İçin Veri Değişikliği



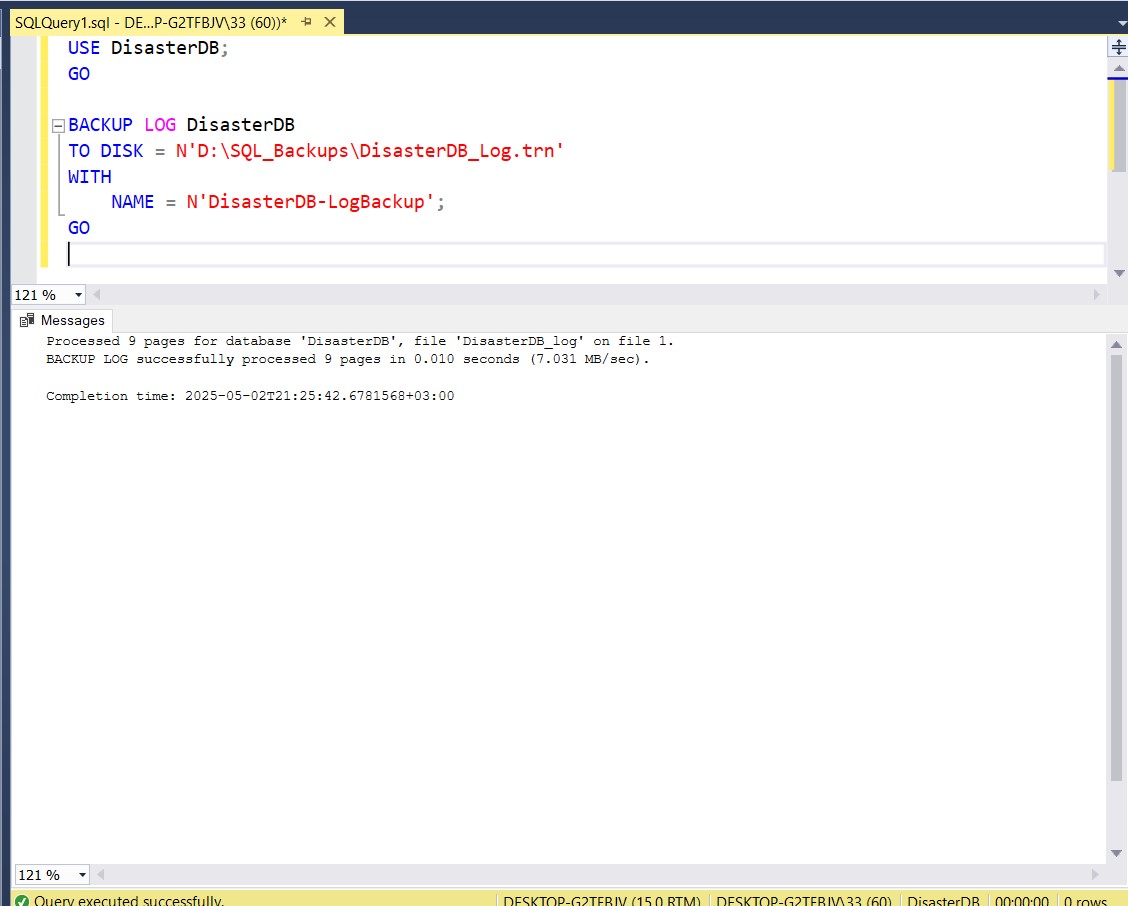
Transaction‑log yedeğini almadan hemen önce veritabanında değişiklik yapıyoruz ki log backup’ı aldığımızda bu eklemenin de log’a işlenmiş olduğunu görebilelim. Felaket kurtarma senaryosunda log yedeğini en güncel değişiklikleri içerecek şekilde almamız gerekiyor.

Adım 3 – Differential Yedek Oluşturma



Bu komutla, son full yedeğin alındığı andan sonra yapılan değişiklikleri içeren “differential” yedeği aldık. Böylece her seferinde tüm veriyi tekrar yedeklemek yerine, sadece değişen sayfalar yedeklenir ve kurtarma süreci hızlanır.

Adım 4 – Transaction Log (Log) Yedeği Oluşturma

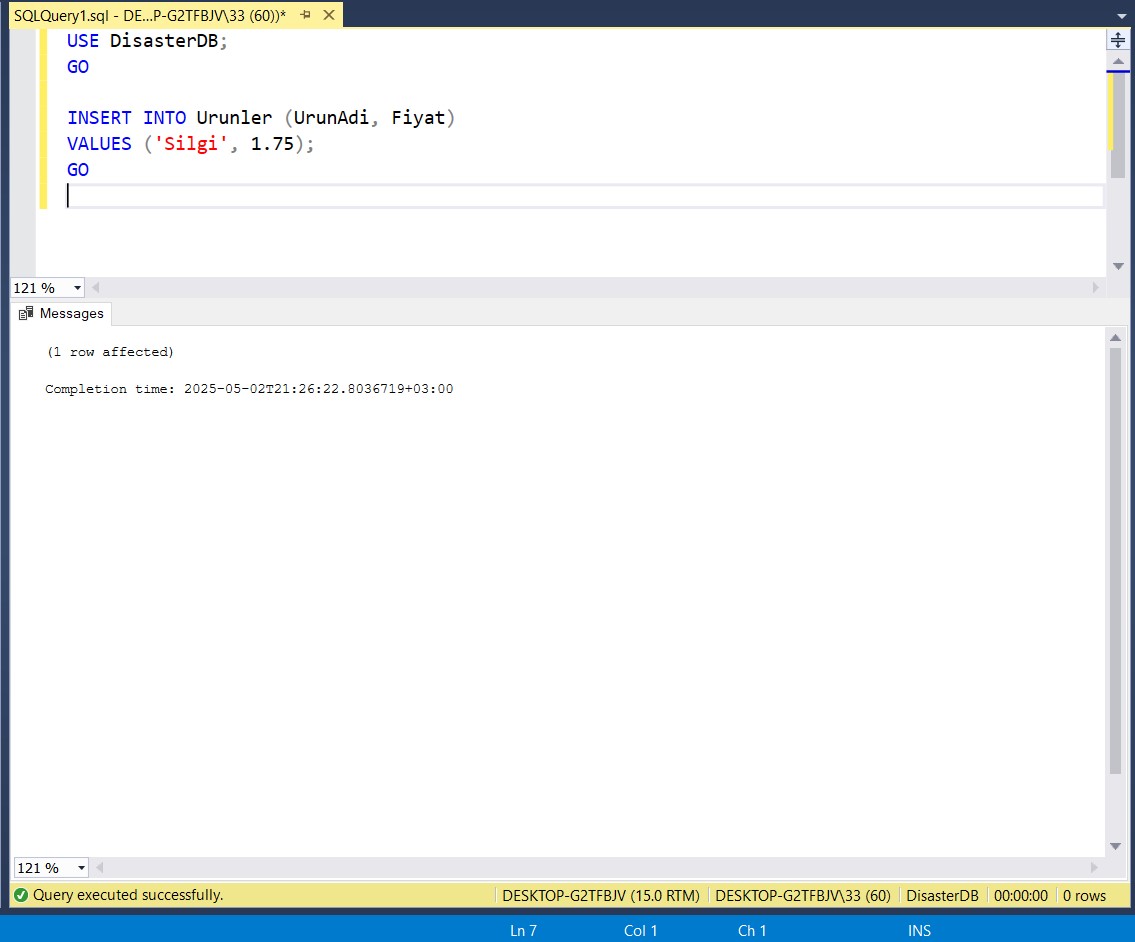


 USE DisasterDB; ile çalıştığımız veritabanını seçtik.

 BACKUP LOG komutuyla, tam ve differential yedekler arasındaki tüm işlem kayıtlarını (transaction log) bir .trn dosyasına kaydettik.

 NAME = N'DisasterDB-LogBackup' parametresiyle yedeğe bir etiket verdik.

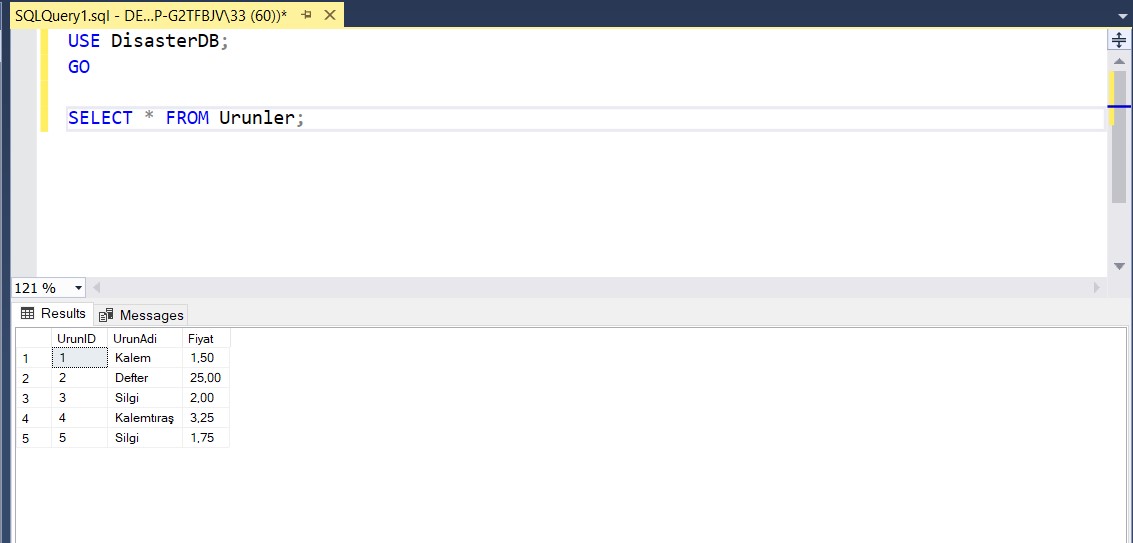
Adım 5 – Point‑in‑Time (Zamana Özel) Kurtarma Senaryosu İçin Ek Veri Üretme



Bu aşamada, daha önce tam ve fark yedekleri alındıktan ve transaction log yedeği alındıktan sonra veritabanına yeni bir kayıt ekleyerek “zamanına özel” kurtarmayı test etmek için bir değişiklik üretiyoruz.

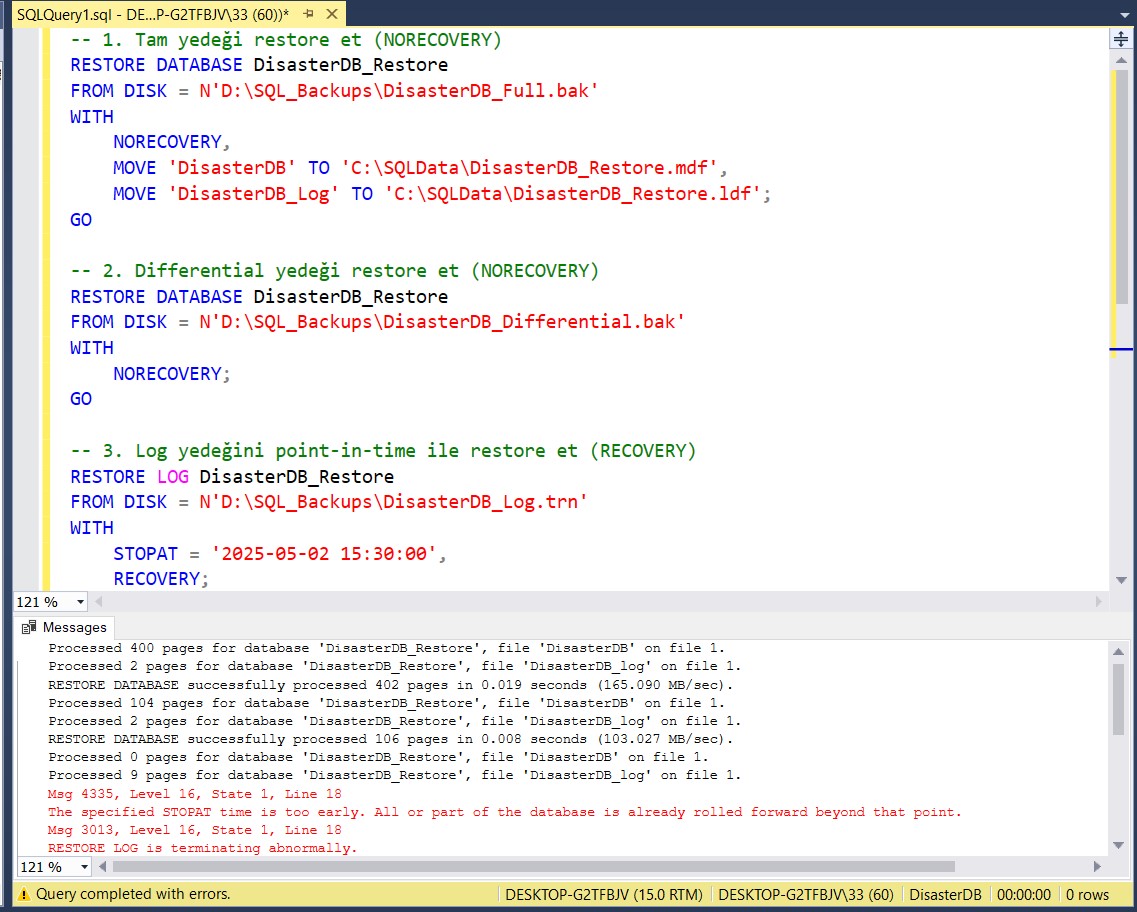
 Tam ve fark yedeklerini elde ettikten sonra sistem üzerinde yeni bir işlem (INSERT) yaptığımızı göstermek, transaction log’daki bu değişikliğin “up to a point” (belirli bir noktaya kadar) kurtarma senaryosunda nasıl atlanabileceğini test etmemize yarar.

 Bir sonraki adımda, RESTORE LOG ... WITH STOPAT = '<süre>'; komutuyla “Silgi” satırının eklenmediği bir zamana geri dönebileceğimizi kanıtlamış olacağız.



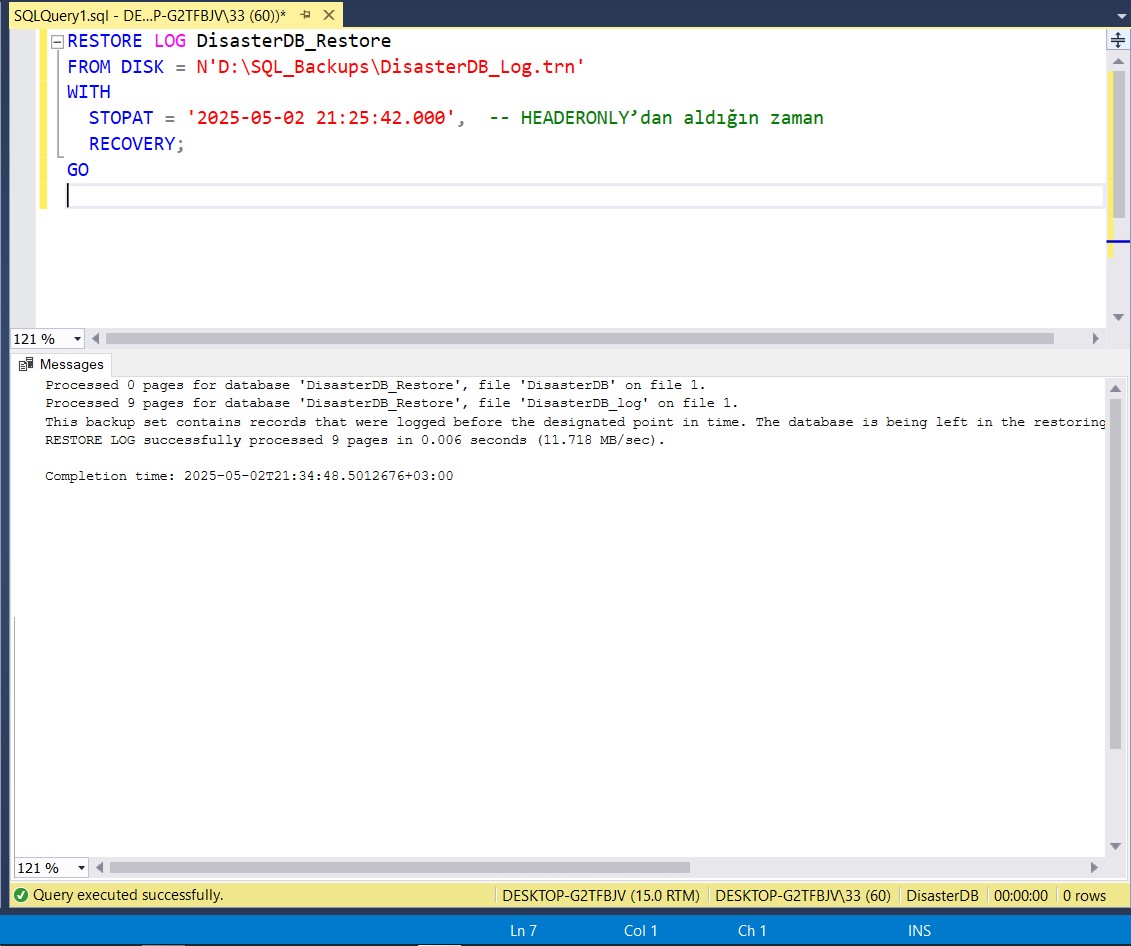
Silgi satırının eklendiğini gösteren sorgu

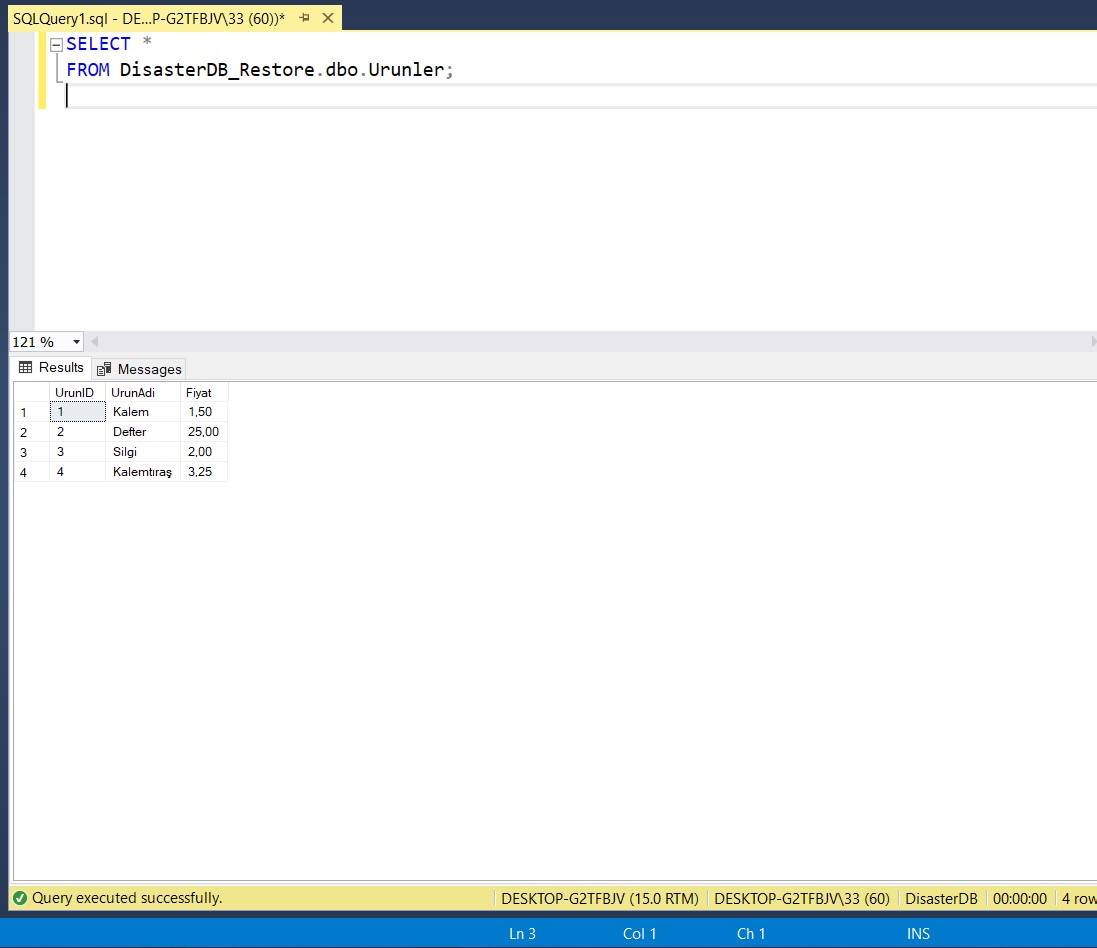
Adım 6 – Point‑in‑Time Kurtarma (Silgi Satırını Geri Atlama)



**RESTORE LOG** komutunun STOPAT = '2025‑05‑02 15:30:00' zamanını işleyemediğini ve “specified STOPAT time is too early” hatası verdiğini görüyoruz.

15:30:00 zamanından **sonraki** işlemler zaten işlenmiş (yani veritabanı log’u o noktaya kadar geri sarılamıyor, çünkü o noktadan önceki tüm işlem kayıtları çoktan uygulanmış).



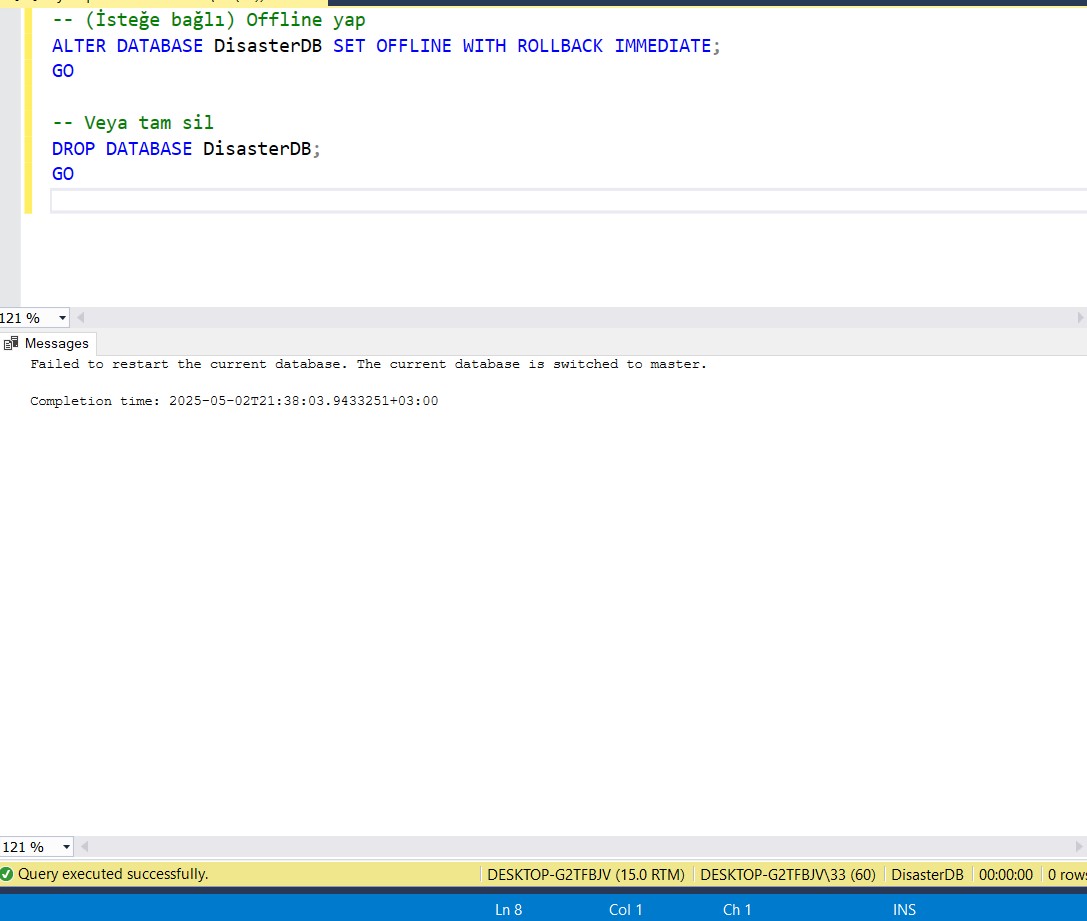


* **Son eklediğim “Silgi” kaydı** (UrunID = 5, fiyat 1.75) ise kurtarmayı durdurduğunuz zamandan **sonra** eklendiği için geri yükleme kapsamına alınmadı ve tabloda **silinmiş** olarak görünüyor.

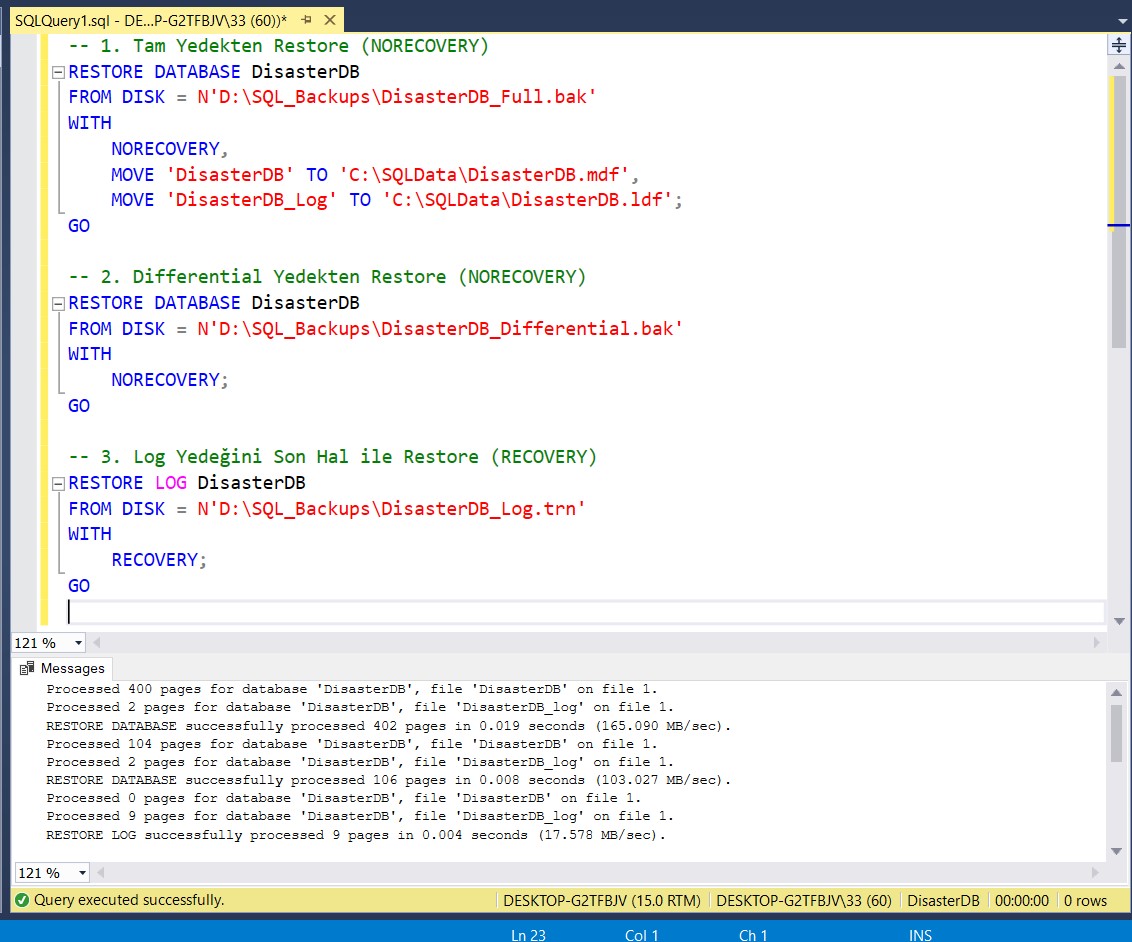
Yani point‑in‑time restore’ı log’daki STOPAT zamanıyla eşleştirdiğinizde, o andan sonra yapılmış ekleme/güncelleme işlemleri geri alınır. Bu sayede istenmeyen bir değişikliği “zaman içinde geri sararak” ortadan kaldırabilirsiniz.

Adım-7 Felaket Senaryosu Testi (DB Kaybından Geri Dönüş)

**7.1 Orijinal Veritabanını Kaldır / Offline Yap**



7.2 Tam + Differential + Log Yedek Zinciriyle Geri Yükleme



7.3 Tüm Veritabanı Geri Geldi

